AN ABSTRACT OF THE THESIS OF

Erkay Savaş for the degree of Doctor of Philosophy

in Electrical & Computer Engineering presented on June 20, 2000.

Title: Implementation Aspects of

Elliptic Curve Cryptography

Redacted for Privacy

Abstract approved: _____

Çetin K. Koç

As the information-processing and telecommunications revolutions now underway will continue to change our life styles in the rest of the 21st century, our personal and economic lives rely more and more on our ability to transact over the electronic medium in a secure way. The privacy, authenticity, and integrity of the information transmitted or stored on networked computers must be maintained at every point of the transaction. Fortunately, cryptography provides algotrithms and techniques for keeping information secret, for determining that the contents of a transaction have not been tampered with, for determining who has really authorized the transaction, and for binding the involved parties with the contents of the transaction. Since we need security on every piece of digital equipment that helps conduct transactions over the internet in the near future, space and time performances of cryptographic algorithms will always remain to be among the most critical aspects of implementing cryptographic functions.

A major class of cryptographic algorithms comprises public-key schemes which enable to realize the message integrity and authenticity check, key distribution, digital signature functions etc. An important category of public-key algorithms is that of elliptic curve cryptosystems (ECC). One of the major advantages of elliptic curve cryptosystems is that they utilize much shorter key lengths in comparison to other well known algorithms such as RSA cryptosystems. However, as do the other

public-key cryptosystems ECC also requires computationally intensive operations. Although the speed remains to be always the primary concern, other design constraints such as memory might be of significant importance for certain constrained platforms.

In this thesis, we are interested in developing space- and time-efficient hardware and software implementations of the elliptic curve cryptosystems. The main focus of this work is to improve and devise algorithms and hardware architectures for arithmetic operations of finite fields used in elliptic curve cryptosystems.

Implementation Aspects of Elliptic Curve Cryptography

by

Erkay Savaş

A THESIS submitted
to
Oregon State University

in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Completed June 20, 2000
Commencement June 2001

Doctor of Philosophy thesis of Erkay Savaş presented on June 20, 2000

APPROVED:

Redacted for Privacy

Major Professor, representing Electrical & Computer Engineering

Redacted for Privacy

Head of Department of Electrical & Computer Engineering

Redacted for Privacy

Dean of Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for Privacy

Erkay Savaş, Author

ACKNOWLEDGMENTS

I am grateful to my advisor, Dr. Çetin K. Koç, not only for his support and advice on academic work and but also for his efforts to create an excellent research enviroment that gives me the opportunity to complete my degree.

I would like to give my special thanks to Dr. Alexandre F. Tenca for his help in the design of scalable and unified multiplier architecture for binary extension and prime fields presented in Chapter 3. The design is based on his previously proposed multiplier architecture.

And special thanks go to Dr. Thomas A. Schmidt for his help, valuable advice and productive discussions in order to devise and implement the algorithm for generating elliptic curves given in Chapter 6. The material included in Chapter 6 for explaining the theory behind the mathematics of so called Complex Multiplication method for generating elliptic curves of known orders is prepared with his expertise in the field, without which I would not succeed in implementing the algorithm.

I would also like to thank Dr. Burt Fein and Dr. Virginia Stonick for their support and understanding as members of my Ph.D. committee.

Thanks also to my colleagues in Information Security Laboratories for their general support and being such good friends.

And finally, I would like to mention my wife, Pınar Tunç, without whose support I could not have succeeded.

Erkay Savaş

Corvallis, Oregon, June 2000

TABLE OF CONTENTS

TABLE OF CONTENTS (CONTINUED)

TABLE OF CONTENTS (CONTINUED)

LIST OF FIGURES

LIST OF TABLES

Çok sevgili babam, Necmettin Savaş'ın değerli anısına

Implementation Aspects of Elliptic Curve Cryptography

Chapter 1
Introduction

As the information-processing and telecommunications revolutions now underway will continue to affect and change our life styles in the 21st century, our personal and economic lives rely more and more on our ability to transact over the electronic medium in a secure way. We have already started to buy goods online, bank online, and conduct business and financial transactions online. Unfortunately, the technology enabling those remote transactions relies on broadcasting everything as sequences of ones and zeroes over an open and insecure channel. The privacy, authenticity, and integrity of the information transmitted or stored on networked computers must be maintained at every point of the transaction. Otherwise, it is impossible for a bank to make sure of the authenticity and/or the integrity of a request from a costumer regarding, for example, a transfer of a certain amount of money to another bank. Similarly, there is no way to distinguish digital money if it is easily counterfeited in the digital environment. Fortunately, cryptography provides techniques for keeping information secret, for determining that the contents of a transaction has not been tampered with, for determining who has really authorized the transaction, and for binding the involved parties with the contents of the transaction.

A major category of cryptographic algorithms comprises public-key schemes which enable to realize the message integrity and authenticity check, key distribution, digital signature etc. Public-key algorithms are based on intractable problems and they require extensive amount of computer resources such as computing power and memory. Especially on relatively slow microprocessors with limited memory, public-key algorithms might be very inefficient if special attention is not paid to certain implementation issues.

An important class of public-key algorithms is that of elliptic curve cryptosystems (ECC) proposed independently by Miller and Koblitz [23, 32]. One of the major advantages of elliptic curve cryptosystems is that they utilize much shorter key lengths in comparison to other well known algorithms such as RSA cryptosystems. In other words, ECC delivers the highest cryptographic strength per bit of any known public-key cryptosystem. A preliminary analysis shows that the elliptic curve digital signature algorithm (ECDSA) with 160-bit key size is equivalent in strength to the RSA digital signature algorithm (RSADSA) with 1024-bit key size. Therefore, ECC enables smaller and faster public-key cryptosystem implementations for even the most constrained environments such as smart cards.

However, as do the other public-key cryptosystems ECC also requires computationally intensive operations. Elementary operations in the algebraic structures (finite fields) over which elliptic curves are constructed play a decisive role in the efficiency of the ECC implementations. Although the speed remains to be always the primary concern, other design constraints such as memory may be of significant importance for certain constrained platforms. Another implementation aspect of the ECC is the generation of secure curves which is considered to be a hard implementation problem. Randomly chosen curves with suitable known orders must be used in elliptic curve cryptosystems.

In this thesis, we are interested in developing space- and time-efficient hardware and software implementations of the elliptic curve cryptosystems. The main objective of our work is to improve and devise algorithms for finite field arithmetic used in elliptic curve cryptosystems for different platforms. We study different representation methods of finite fields proposed for efficient implementations and determine the best arithmetic algorithms for these representations. Methodologies such as precomputation and utilization of acceleration tables when memory is available are adopted to obtain efficient algorithms.

Special emphasis is given to scalability of the algorithms and hardware architectures for finite field arithmetic to longer precision as the growing need for

higher security necessitates longer key sizes. We are not interested in the inflexible solutions which provides efficient methods for only a small class of key sizes.

We prefer to work on finite fields of prime characteristic $GF(p)$ and binary extension fields $GF(2^k)$, mainly because they are prevalently used in cryptographic schemes and we are more familiar with their structural properties.

We are especially interested in Montgomery arithmetic previously proposed for both fields. The well-known Montgomery multiplication algorithm enables very efficient software and hardware implementations. The classic method we use to perform multiplication operation in $GF(p)$ and $GF(2^k)$ consists of two steps : (1) regular integer or polynomial multiplication and (2) division. Depending of the prime number chosen for $GF(p)$ or irreducible polynomial for $GF(2^k)$, the second step might be extremely time-comsuming. One way to circumvent this problem is to use special primes and irreducible polynomials. The Montgomery multiplication, on the other hand, provides the same performance regardless of the prime or irreducible polynomial of the field. It basically replaces the division operation in the second step of the classic algorithm with extra multiplications and shift operations. Since the division is almost always much costlier than the multiplication, the Montgomery multiplication algorithm is beneficial especially for randomly chosen primes and irreducible polynomials. Even though faster algorithms were proposed for special primes and irreducible polynomials, working with random primes and polynomials provides extra flexibility and wider collection of finite fields.

Beside the Montgomery multiplication algorithms we need algorithms for inversion in order to do Montgomery arithmetic in a given field. The existing inversion algorithms need to be modified to calculate the Montgomery inversion of a given field element. We give special consideration to a proper definition of the Montgomery inversion which has not been given in previous works. We also provide an efficient algorithm to compute the Montgomery inversion.

The composite representation of binary extension fields provides very efficient software implementations for field arithmetic thanks to the utilization of tables

and wordwise algorithms. Employment of the composite fields necessitates the efficient conversion algorithms between composite fields and the other representation methods.

The use of Montgomery arithmetic helps reduce the complexity of elliptic curve generation algorithms by enabling the flexibility of selecting prime number for $GF(p)$. In this thesis, we introduce the fundementals of so called *complex multiplication* method for elliptic curve generation and the simplifications of the method for $GF(p)$ when we have the flexibility of selecting the characteristic of the field. The thesis is organized as follows:

In Chapter 2, we modify an algorithm given by Kaliski to compute the Montgomery inverse of an integer modulo a prime number. We also give a new definition of the Montgomery inverse, and introduce efficient algorithms for computing the classical modular inverse, the Kaliski-Montgomery inverse, and the new Montgomery inverse. The proposed algorithms are suitable for software implementations on general-purpose microprocessors.

In chapter 3, we describe a scalable and unified architecture for a Montgomery multiplication module which operates in both types of finite fields $GF(p)$ and $GF(2^m)$. The unified architecture requires only slightly more area than that of the multiplier architecture for the field $GF(p)$. The multiplier is scalable, which means that a fixed-area multiplication module can handle operands of any size, and also, the wordsize can be selected based on the area and performance requirements. We utilize the concurrency in the Montgomery multiplication operation by employing a pipelining design methodology. We also describe a scalable and unified adder module to carry out concomitant operations in our implementation of the Montgomery multiplication. The upper limit on the precision of the scalable and unified Montgomery multiplier is dictated only by the available memory to store the operands and internal results, and the module is capable of performing infinite-precision Montgomery multiplication in both types of finite fields.

In Chapter 4, we propose new and efficient algorithms for obtaining software implementations of the basic arithmetic (squaring, multiplication, and inversion) operations in the Galois fields $GF(2^k)$ where k is a composite integer as $k = nm$. These algorithms are suitable for obtaining software implementations of the field operations in microprocessors and signal processors, and they are particularly useful for applications in public-key cryptography where the field size is large.

Chapter 5 describes a method of construction of a composite field $GF((2^n)^m)$ given the binary field $GF(2^k)$, where $k = nm$. We also derive the conversion (change of basis) matrix from $GF((2^n)^m)$ to $GF(2^k)$. The special case of $\gcd(n, m) = 1$, where the irreducible polynomial generating the field $GF((2^n)^m)$ has its coefficients from $GF(2)$ rather than $GF(2^n)$, is also treated. Furthermore, certain generalizations of the proposed construction method, e.g., the use of non-primitive elements and the construction of composite fields with special irreducible polynomials, are also discussed. Finally, we give storage-efficient conversion algorithms between the binary and composite fields for the case $\gcd(n, m) = 1$.

In Chapter 6, we give a methodology to generate suitable elliptic curves over $GF(p)$. Our method is based on the *Complex Multiplication* (*CM*) technique. A previously proposed method which is also based on the CM technique assumes that the characteristic p of $GF(p)$ is fixed. However, there are numerous primes within the range of cryptographic interest and flexibility in selecting a prime as the characteristic of the field allows simplification of the complex multiplication curve generation algorithm. Based on this observation, we modify the existing algorithm and provide performance results of our new algorithm. With this modified algorithm, we can utilize prime numbers in certain subsets of all prime numbers. Theoretical analysis and experimental figures show that there are sufficiently many primes in this set so that it is always possible to find such primes after several trials. We also provide experimental results on the plentitude of elliptic curves. The software implementation of the modified method is faster, easier and smaller.

Chapter 2
The Montgomery Modular Inverse - Revisited

2.1 Introduction

The basic arithmetic operations (i.e., addition, multiplication, and inversion) modulo a prime number p have several applications in cryptography, for example, the decipherment operation in the RSA algorithm [44], the Diffie-Hellman key exchange algorithm [8], the US Government Digital Signature Standard [36], and also recently elliptic curve cryptography [23, 30]. The modular inversion operation plays an important role in public-key cryptography, particularly, to accelerate the exponentiation operation using the so-called addition-subtraction chains [9, 22] and also in computing point operations on an elliptic curve defined over the finite field $GF(p)$ [23, 30].

The modular inverse of an integer $a \in [1, p-1]$ modulo the prime p is defined as the integer $x \in [1, p-1]$ such that $ax = 1 \pmod{p}$. It is often written as $x = a^{-1} \pmod{p}$. This is the classical definition of the modular inverse [22]. In the sequel, we will use the notation

$$x := \text{ModInv}(a) = a^{-1} \pmod{p} \tag{2.1}$$

to denote the inverse of a modulo p. The definition of the modular inverse was recently extended by Kaliski to include the so-called Montgomery inverse [18] based on the Montgomery multiplication algorithm [34]. In this chapter, we introduce a new definition of the Montgomery inverse, and also give efficient algorithms to compute the classical modular inverse, the Kaliski-Montgomery inverse, and the new Montgomery inverse of an integer a modulo the prime number p.

2.2 The Montgomery Inverse

The Montgomery multiplication [34] of two integers $a, b \in [0, p-1]$ is defined as $c = ab2^{-n} \pmod{p}$ where $n = \lceil \log_2 p \rceil$. We denote this multiplication operation using the notation

$$c := \text{MonPro}(a, b) = ab2^{-n} \pmod{p}\ , \tag{2.2}$$

where p is the prime number and n is its bit-length. The Montgomery inverse of an integer $a \in [1, p-1]$ is defined by Kaliski [18] as the integer $x = a^{-1}2^n \pmod{p}$. Similarly, we will use the notation

$$x := \text{MonInv}(a) = a^{-1}2^n \pmod{p} \tag{2.3}$$

to denote the Montgomery inversion as defined by Kaliski. The algorithm introduced in [18] computes the Montgomery inverse of a. We give this algorithm below. The output of Phase I is the integer r such that $r = a^{-1}2^k \pmod{p}$, where $n \leq k \leq 2n$. This result is then corrected using Phase II to obtain the Montgomery inverse $x = a^{-1}2^n \pmod{p}$.

Phase I

Input: $a \in [1, p-1]$ and p

Output: $r \in [1, p-1]$ and k,
where $r = a^{-1}2^k \pmod{p}$ and $n \leq k \leq 2n$

1: $u := p$, $v := a$, $r := 0$, and $s := 1$
2: $k := 0$
3: while $(v > 0)$
4: if u is even then $u := u/2$, $s := 2s$
5: else if v is even then $v := v/2$, $r := 2r$
6: else if $u > v$ then $u := (u - v)/2$, $r := r + s$, $s := 2s$
7: else if $v \geq u$ then $v := (v - u)/2$, $s := s + r$, $r := 2r$
8: $k := k + 1$

9: if $r \geq p$ then $r := r - p$

10: return $r := p - r$ and k

Phase II

Input: $r \in [1, p-1]$, p, and k from Phase I

Output: $x \in [1, p-1]$, where $x = a^{-1}2^n \pmod{p}$

11: for $i = 1$ to $k - n$ do

12: if r is even then $r := r/2$

13: else then $r := (r + p)/2$

13: return $x := r$

2.3 The Almost Montgomery Inverse

As shown above, Phase I computes an integer $r = a^{-1}2^k \pmod{p}$, where $n \leq k \leq 2n$. The Montgomery inverse of a is defined as $x = a^{-1}2^n \pmod{p}$ where $n = \lceil \log_2 p \rceil$. We will call the output of the Phase I the *almost Montgomery inverse* of a, and denote it as

$$(r, k) := \text{AlmMonInv}(a) = a^{-1}2^k \pmod{p}\ , \tag{2.4}$$

where $n \leq k \leq 2n$, in the sequel. We note that a similar concept, the almost inverse of elements in the Galois field $GF(2^m)$, was introduced in [48] and some implementation issues were addressed in [45].

Since k is an output of the Phase I, we will include it in the definition of the AlmMonInv function as an output value. We also propose to make an additional change in the way the almost Montgomery inverse algorithm is being used. Instead of selecting the Montgomery radix as $R = 2^n$ where $n = \lceil \log_2 p \rceil$, we will select it as $R = 2^m$, where m is an integer multiple of the wordsize of the computer w, i.e., $m = iw$ for some positive integer i. The Montgomery product algorithm would work with any m as long as $m \geq n$, where n is the bit-length of the prime

number p. For efficiency reasons, we select the smallest i which makes m larger than n, in other words, $iw = m \geq n$, but $(i-1)w < n$. It turns out that the almost Montgomery inverse algorithm (Phase I) works for this case as well. Furthermore, it even works for an input a which may be larger than p as long as it is less than 2^m, as proven below in Theorem 1. The second issue is the value of k after the almost Montgomery inverse algorithm terminates. We show in Theorem 2 below that $n \leq k \leq m + n$.

Theorem 2.1 *If $p > 2$ is a prime and $a \geq 1$ (a might be larger than p), then the intermediate values r, s, and u in the almost Montgomery inverse algorithm are always in the interval $[0, 2p - 1]$.*

Proof If $a < p$, then the proof given in [18] is applicable here. If $a > p$ and a is not an integer multiple of p, then only Step 5 and Step 7 are executed in the while loop until v becomes smaller than u. Until then, the variables u, r, and s keep their initial values. They start changing when $v < u$, and after this point, the algorithm proceeds as in the case $a < p$. Thus, the intermediate values remain in the interval $[0, 2p - 1]$ for $a > p$ as well. □

Theorem 2.2 *If $p > 2$ is a prime and $a \geq 1$, then the index k produced at the end of the almost Montgomery inverse algorithm takes a value between n and $m + n$, where $n = \lceil \log_2 p \rceil$ and $m = sw$ with $sw \geq n$ with $(s-1)w < n$.*

Proof The reduction of uv and $u+v$ at each iteration (at Steps 4-7) is illustrated below:

	uv	$u+v$
Step 4	$(uv)/2$	$(u+2v)/2$
Step 5	$(uv)/2$	$(2u+v)/2$
Step 6	$u^2/2 - (uv)/2$	$(u+v)/2$
Step 7	$(uv)/2 - v^2/2$	$(u+v)/2$

Note that these steps are mutually exclusive, i.e., at an iteration only one of the four cases occurs. At each iteration, the value uv is at least halved while the value $u + v$ is at most halved, and furthermore, both u and v are equal to 1 before the last iteration. Since the initial values of the product uv and the sum $u + v$ are ap and $a + p$, respectively, the index value k (i.e., the number of iterations) satisfies

$$(a + p)/2 \leq 2^{k-1} \leq ap \ .$$

Since $2^{n-1} < p < 2^n$ and $0 < a < 2^m$, we have

$$\begin{aligned} 2^{n-2} < \ & 2^{k-1} \ < 2^m \cdot 2^n \ , \\ 2^{n-1} \leq \ & 2^{k-1} \ \leq 2^{m+n-1} \ . \end{aligned}$$

Thus, we obtain the result: $n \leq k \leq m + n$. Furthermore, we note that $m - n \leq w - 1$, where w is the word size of the machine. This implies that $m - w + 1 \leq k \leq m + n$. □

2.4 Using the Almost Montgomery Inverse

The Montgomery inverse algorithm computes $x = a^{-1}2^n \pmod{p}$. The Kaliski algorithm [18] uses the bit-level operations in Phase II in order to achieve its goal. It uses $k - n$ steps in Phase II, where at each step a bit-level right shift operation is performed. Additionally, if r is odd, an addition operation $r + p$ needs to be performed.

As suggested earlier, we will use the definition $x = a^{-1}2^m \pmod{p}$. Furthermore, it is possible to eliminate the bit-level operations completely, and use the Montgomery product algorithm to obtain the same result. In our approach, we replace these bit-level operations by word-level Montgomery product operations which are intrinsically faster on microprocessors, particularly when the wordsize of the computer is large (i.e., 16, 32, or 64).

The new Phase II is based on the pre-computed Montgomery radix $R = 2^m \pmod{p}$, however, we only need $R^2 \pmod{p}$. This value can be pre-computed and saved, and used as necessary. Another issue is the range of input variables to the AlmMonInv and MonPro functions. For both of these functions, any input cannot exceed $2^m - 1$.

2.4.1 The Modified Kaliski-Montgomery Inverse

This algorithm computes $x = \text{MonInv}(a) = a^{-1}2^m \pmod{p}$ given the integer a. Thus, it finds the inverse of the integer a modulo p and also converts it to the Montgomery domain. The modified Kaliski-Montgomery inverse algorithm is given below.

Input: a, p, n, and m, where $a \in [1, 2^m - 1]$.
Output: $x = a^{-1}2^m \pmod{p}$, where $x \in [1, p-1]$.

1: $(r, k) := \text{AlmMonInv}(a)$ where $r = a^{-1}2^k \pmod{p}$
and $n \leq k \leq m + n$.
2: If $n \leq k \leq m$ then
2.1: $r := \text{MonPro}(r, R^2) = (a^{-1}2^k)(2^{2m})(2^{-m}) =$
$a^{-1}2^{m+k} \pmod{p}$
2.2: $k := k + m > m$
3: $r := \text{MonPro}(r, 2^{2m-k}) = a^{-1} \cdot 2^k \cdot 2^{2m-k} \cdot 2^{-m} =$
$a^{-1}2^m \pmod{p}$
4: Return $x = r$, where $x = a^{-1}2^m \pmod{p}$

The inputs to the MonPro function in Step 2.1 are r and R^2, which are both in the correct range. The input 2^{2m-k} to MonPro in Step 3 is also in the correct range since k is adjusted to be larger than m in Step 2.2 when $k \leq m$, thus, $0 < 2^{2m-k} < 2^m$.

2.4.2 The Classical Modular Inverse

In some cases, we are only interested in computing $x = \text{ModInv}(a) = a^{-1} \pmod{p}$ without converting to the Montgomery domain. One way to achieve this is first to compute the Kaliski-Montgomery inverse of a to obtain $b = a^{-1}2^m \pmod{p}$, and then re-convert the result back to the residue (non-Montgomery) domain using the Montgomery product as

$$\begin{aligned} b &:= \text{MonInv}(a) = a^{-1}2^m \bmod p \ , \\ x &:= \text{MonPro}(b, 1) = (a^{-1}2^m)(1)(2^{-m}) = a^{-1} \bmod p \ . \end{aligned}$$

Another way of computing the classical inverse is by reversing the order of MonInv and MonPro operations, and using the constant $R^2 = 2^{2m} \pmod{p}$ as follows:

$$\begin{aligned} b &:= \text{MonPro}(a, R^2) = (a)(2^{2m})(2^{-m}) = a2^m \bmod p \ , \\ x &:= \text{MonInv}(b) = (a2^m)^{-1}2^m = a^{-1} \bmod p \ . \end{aligned}$$

However, either one of these approaches requires 2 or 3 Montgomery product operations in addition to the AlmMonInv function. Instead, we can modify the Kaliski-Montgomery inverse algorithm so that it directly computes the classical modular inverse after the AlmMonInv function with 1 or 2 Montgomery product operations.

Input: a, p, n, and m, where $a \in [1, 2^m - 1]$

Output: $x = a^{-1} \pmod{p}$, where $x \in [1, p - 1]$

1: $(r, k) := \text{AlmMonInv}(a)$ where $r = a^{-1}2^k \pmod{p}$
and $n \leq k \leq m + n$.

2: If $k > m$ then

2.1: $r := \text{MonPro}(r, 1) = (a^{-1}2^k)(2^{-m}) =$
$a^{-1}2^{k-m} \pmod{p}$

2.2: $k := k - m < m$

3: $r := \text{MonPro}(r, 2^{m-k}) = (a^{-1})(2^k)(2^{m-k})(2^{-m}) =$
$a^{-1} \pmod{p}$

4: Return $x = r$, where $x = a^{-1} \pmod{p}$

2.4.3 The New Montgomery Inverse

We propose the following new definition of the Montgomery inverse: $x = a^{-1}2^{2m} \pmod{p}$ given the input $a \pmod{p}$. According to this new definition, we compute the Montgomery inverse of an integer which is already in the Montgomery domain, producing the output x which is also in the Montgomery domain. We will denote the new Montgomery inverse computation by

$$x := \text{NewMonInv}(a2^m) = (a2^m)^{-1}2^{2m} = a^{-1}2^m \bmod p \ .$$

The Kaliski-Montgomery inverse of a is defined as $\text{MonInv}(a) = a^{-1}2^m \pmod{p}$, which has the following property

$$\begin{aligned} \text{MonPro}(a, \text{MonInv}(a)) &= \text{MonPro}(a, a^{-1}2^m) \\ &= a(a^{-1}2^m)2^{-m} = 1 \pmod{p} \ . \end{aligned}$$

In other words, according to the Kaliski-Montgomery inverse, the multiplicative identity is equal to 1, which is an incorrect assumption if we are operating in the Montgomery domain where the image of 1 is $2^m \pmod{p}$. On the other hand, the new Montgomery inverse has the following property

$$\begin{aligned} \text{MonPro}(a2^m, \text{NewMonInv}(a2^m)) &= (a2^m)(a^{-1}2^m)2^{-m} \\ &= 2^m \pmod{p} \ . \end{aligned}$$

This new definition of the inverse is more suitable for computing expressions using the Montgomery multiplication since it computes the result in the Montgomery domain.

The new Montgomery inverse cannot be directly computed using the MonInv algorithm by giving the input as $a2^m \pmod{p}$, since we would obtain

$$\text{MonInv}(a2^m) = (a2^m)^{-1}2^m = a^{-1}2^{-m}2^m = a^{-1} \bmod p \ .$$

However, this can be converted back to the Montgomery domain using a single Montgomery product with $R^2 \pmod{p}$. Thus, we obtain a method of computing

the new Montgomery inverse as

$$\begin{aligned} b &:= \text{MonInv}(a2^m) = (a2^m)^{-1}2^m = a^{-1} \pmod{p} , \\ x &:= \text{MonPro}(b, R^2) = a^{-1}(2^{2m})2^{-m} = a^{-1}2^m \bmod p . \end{aligned}$$

Similarly, another method to obtain the same result is by reversing the order of the operations:

$$\begin{aligned} b &:= \text{MonPro}(a2^m, 1) = (a2^m)(1)(2^{-m}) = a \bmod p , \\ x &:= \text{MonInv}(a) = a^{-1}2^m \bmod p . \end{aligned}$$

The new algorithm uses the pre-computed value $R^2 \pmod{p}$, and it is more efficient: It uses only 2 or 3 Montgomery product operations after the AlmMonInv function.

Input: $a2^m \pmod{p}$, p, n, and m
Output: $x = a^{-1}2^m \pmod{p}$, where $x \in [1, p-1]$

1: $(r, k) := \text{AlmMonInv}(a2^m)$ where
$r = a^{-1}2^{-m}2^k \pmod{p}$ and $n \leq k \leq m + n$
2: If $n \leq k \leq m$ then
2.1: $r := \text{MonPro}(r, R^2) = (a^{-1}2^{-m}2^k)(2^{2m})(2^{-m}) =$
$a^{-1}2^k \pmod{p}$
2.2: $k := k + m > m$
3: $r := \text{MonPro}(r, R^2) = (a^{-1}2^{-m}2^k)(2^{2m})(2^{-m}) =$
$a^{-1}2^k \pmod{p}$
4: $r := \text{MonPro}(r, 2^{2m-k}) = (a^{-1}2^k)(2^{2m-k})(2^{-m}) =$
$a^{-1}2^m \pmod{p}$
5: Return $x = r$, where $x = a^{-1}2^m \pmod{p}$

2.5 Conclusions and Applications

We have proposed a new definition of the Montgomery inverse, and have given efficient algorithms to compute the classical modular inverse, the Kaliski-Montgomery inverse, and the new Montgomery inverse. The new algorithms are based on the almost Montgomery inverse function and require 2 or 3 Montgomery product operations thereafter, instead of using the bit-level operations as in [18].

Table 2.1. Implementation results

Bit	PhI	Algorithm	Old PhII	New PhII	PhII Spd	All Spd
160	138 μs	MonInv	30 μs	9 μs	3.34	1.14
		ModInv	85 μs	10 μs	8.50	1.51
		NewMonInv	57 μs	10 μs	5.70	1.32
192	192 μs	MonInv	39 μs	11 μs	3.54	1.14
		ModInv	128 μs	13 μs	9.85	1.56
		NewMonInv	87 μs	13 μs	6.69	1.36

We have performed some experiments by implementing all three inversion algorithms using both classical (shift and add) and newly proposed Montgomery product based Phase II steps. These algorithms were coded using Microsoft Visual C++ 5.0 development system. The timing results are obtained on a 450-MHz Pentium II processor running the Windows NT 4.0 operating system. In Table 2.1, we summarize the timing results. The table contains the old and new Phase II timings (Old PhII and New PhII) in microseconds for operands of length 160 and 192 bits. The last two columns (PhII Spd and All Spd) give the speedup in Phase II only and the overall speedup, which illustrates the efficiency of the algorithms introduced.

An application of the new Montgomery inverse is found in computing eP, where e is an integer and P is a point on an elliptic curve defined over the finite field $GF(p)$. This computation requires that we perform elliptic curve point addition $P+Q$ and doubling $P+P=2P$ operations, where each point operation requires a few modular additions and multiplications, and a modular inversion. The inverse operation is used to compute the variable $\lambda := (y_2 - y_1)(x_2 - x_1)^{-1} \pmod{p}$, which is required in computing elliptic curve point addition of $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ in order to obtain $P + Q = (x_3, y_3)$. Assuming the input variables are given in the Montgomery domain, we would like to obtain the result in the Montgomery domain. If the Kaliski-Montgomery inverse is used, it will compute the classical inverse which is in the residue (non-Montgomery) domain, and cannot be readily used in subsequent operations. We need to perform a Montgomery product with $R^2 \pmod{p}$ in order to convert back to the Montgomery domain. However, with the help of the new Montgomery inverse, we can perform the above computation in a single step. Since these operations are performed for every bit of the exponent e, the new Montgomery inverse is more efficient and highly useful in this context.

Chapter 3
A Scalable and Unified Multiplier Architecture for Finite Fields $GF(p)$ and $GF(2^m)$

3.1 Introduction

The basic arithmetic operations (i.e., addition, multiplication, and inversion) in prime and binary extension fields, $GF(p)$ and $GF(2^m)$, have several applications in cryptography, such as decipherment operation of RSA algorithm [44], Diffie-Hellman key exchange algorithm [8], elliptic curve cryptography [23, 30], and the Digital Signature Standard including the Elliptic Curve Digital Signature Algorithm [37]. The most important of these three arithmetic operations is the field multiplication operation since the it is the core operation in many cryptographic functions.

The Montgomery multiplication algorithm [34] is an efficient method for doing modular multiplication with an odd modulus. The Montgomery multiplication algorithm is a very useful for obtaining fast software implementations of the multiplication operation in prime fields $GF(p)$. The algorithm replaces division operation with simple shifts, which are particularly suitable for implementation on general-purpose computers. The Montgomery multiplication operation has been extended to the finite field $GF(2^k)$ in [25]. Efficient software implementations of the multiplication operation in $GF(2^k)$ can be obtained using this algorithm, particularly when the irreducible polynomial generating the field is chosen arbitrarily. The main idea of the architecture proposed in this chapter is based on the observation that the Montgomery multiplication algorithm for both fields $GF(p)$ and $GF(2^k)$ are essentially the same algorithm. The proposed unified architecture performs the Montgomery multiplication in the field $GF(p)$ generated by an arbitrary prime p

and in the field $GF(2^m)$ generated by an arbitrary irreducible polynomial $p(x)$. We show that a unified multiplier performing the Montgomery multiplication operation in the fields $GF(p)$ and $GF(2^k)$ can be designed at a cost only slightly higher than the multiplier for the field $GF(p)$, providing significant savings when both types of multipliers are needed.

Several variants of the Montgomery multiplication algorithm [40, 26, 3] have been proposed to obtain more efficient software implementations on specific processors. Various hardware implementations of the Montgomery multiplication algorithm for limited precision operands are also reported [3, 40, 10]. On the other hand, implementations utilizing high-radix modular multipliers have also been proposed [40, 28, 46]. Advantages and disadvantages of using high-radix representation have been discussed in [56, 55]. Because high-radix Montgomery multiplication designs introduce longer critical paths and more complex circuitry, these designs are less attractive for hardware implementations.

A scalable Montgomery multiplier design methodology for $GF(p)$ was introduced in [55] in order to obtain hardware implementations. This design methodology allows to use a fixed-area modular multiplication circuit for performing multiplication of unlimited precision operands. The design tradeoffs for best performance in a limited chip area were also analyzed in [55]. We use the design approach as in [55] to obtain a scalable hardware module. Furthermore, the scalable multiplier described in this chapter is capable of performing multiplication in both types finite fields $GF(p)$ and $GF(2^k)$, i.e., it is a scalable and unified multiplier.

The main contributions of this chapter are summarized below.

- We show that a unified architecture for multiplication module which operates both in $GF(p)$ and $GF(2^m)$ can be designed easily without compromising scalability, time and area efficiency.

- We analyze the design considerations such as the effect of word length, the number of the pipeline stages, and the chip area, etc., by supplying implementation results obtained by Mentor graphics synthesis tools.

- We describe the design of a dual-field, scalable adder circuit which is suitable for the pipeline organization of the multiplier. This adder is necessary for the final reduction step in the Montgomery algorithm and in the final addition for converting the result of the multiplication operation (which is in the Carry-Save form) to the nonredundant form. Naturally, the adder operates both in $GF(p)$ and $GF(2^m)$. We give an analysis of the time and area cost of the adder circuit.

We start with a short discussion of scalability in §3.2 and explain the main idea behind the unified multiplier architecture in §3.3. We then present the methodology to perform the Montgomery multiplication operation in both types of finite fields using the unified architecture. We give the original and modified definitions of Montgomery algorithm for $GF(p)$ and $GF(2^m)$ in §3.4. We discuss concurrency in the Montgomery multiplication and show the methodology to design a pipeline module utilizing the concurrency in §3.5. We present the processing unit and the modifications needed to make the unit operate in prime and binary extension fields in §3.6. We then provide a multi-purpose word adder/subtractor module in §3.7, which can be integrated into the main Montgomery multiplier module in order to perform the field addition and subtraction operations. In §3.8, we discuss the area/time tradeoffs and suitable choices for word lengths, the number of pipeline stages, and typical chip area requirements. Finally, we summarize our conclusions in §3.9.

3.2 Scalable Multiplier Architecture

An arithmetic unit is called scalable if it can be reused or replicated in order to generate long-precision results independently of the data path precision for which the unit was originally designed. To speed up the multiplication operation, various dedicated multiplier modules were developed in [46, 1, 35]. These designs operate over a fixed finite field. For example, the multiplier designed for 155 bits [1] cannot be used for any other field of higher degree. When a need for a multiplication

of larger precision arises, a new multiplier must be designed. Another way to avoid redesigning the module is to use software implementations and fixed precision multipliers. However, software implementations are inefficient in utilizing inherent concurrency of the multiplication because of the inconvenient pipeline structure of the microprocessors being used. Furthermore, software implementations on fixed digit multipliers are more complex and require excessive amount of effort in coding. Therefore, a scalable hardware module specifically tailored to take advantage of the concurrency of the Montgomery multiplication algorithm becomes extremely attractive.

3.3 Unified Multiplier Architecture

Even though prime and binary extension fields, $GF(p)$ and $GF(2^m)$, have dissimilar properties, the elements of either field are represented using almost the same data structures inside the computer. In addition, the algorithms for basic arithmetic operations in both fields have structural similarities allowing a unified module design methodology. For example, the steps of the Montgomery multiplication algorithm for binary extension field $GF(2^m)$ given in [25] only slightly differ from those of the integer Montgomery multiplication algorithm [34, 26]. Therefore, a scalable arithmetic module, which can be adjusted to operate in both types of fields, is feasible, provided that this extra functionality does not lead to an excessive increase in area or a dramatic decrease in speed. In addition, designing such a module must require only a small amount of extra effort and no major modification in control logic of the circuit.

Considering the amount of time, money and effort that must be invested in designing a multiplier module or more generally speaking a cryptographic coprocessor, a scalable and unified architecture which can perform arithmetic in two commonly used algebraic fields is definitely beneficial. In this chapter, we show the method to design a Montgomery multiplier that can be used for both types of fields following the design methodology presented in [55]. The proposed unified architecture

is obtained from the scalable architecture given in [55] after minor modifications. The propagation time is unaffected and the increase in chip area is insignificant.

3.4 Montgomery Multiplication

Given two integers A and B, and the prime modulus p, the Montgomery multiplication algorithm computes

$$C = \mathsf{MonMul}(A, B) = A \cdot B \cdot R^{-1} \pmod{p} , \tag{3.1}$$

where $R = 2^m$ and $A, B < p < R$, and p is an m-bit number. The original algorithm works for any modulus n provided that $\gcd(n, R) = 1$. In this chapter, we assume that the modulus is a prime number, thus, we perform multiplication in the field defined by this prime number. This issue is also relevant when the algorithm is defined for the binary extension fields.

The Montgomery multiplication algorithm relies on a different representation of the finite field elements. The field element $A \in GF(p)$ is transformed into another element $\bar{A} \in GF(p)$ using the formula $\bar{A} = A \cdot R \pmod{p}$. The number $\bar{A}$ is called Montgomery image of the element, or $\bar{A}$ is said to be in the Montgomery domain. Given two elements in the Montgomery domain $\bar{A}$ and $\bar{B}$, the Montgomery multiplication computes

$$\bar{C} = \bar{A} \cdot \bar{B} \cdot R^{-1} \pmod{p} = (A \cdot R) \cdot (B \cdot R) \cdot R^{-1} \pmod{p} = C \cdot R \pmod{p} , \tag{3.2}$$

where $\bar{C}$ is again in the Montgomery domain. The transformation operations between the two domains can also be performed using the MonMul function as

$$\begin{aligned} \bar{A} &= \mathsf{MonMul}(A, R^2) = A \cdot R^2 \cdot R^{-1} = A \cdot R \pmod{p} , \\ \bar{B} &= \mathsf{MonMul}(B, R^2) = B \cdot R^2 \cdot R^{-1} = B \cdot R \pmod{p} , \\ C &= \mathsf{MonMul}(\bar{C}, 1) = C \cdot R \cdot R^{-1} = C \pmod{p} . \end{aligned}$$

Provided that $R^2 \pmod{p}$ is precomputed and saved, we need only a single MonMul operation to carry out each of these transformations. However, because

of these transformation operations, performing a single modular multiplication using MonMul might not be advantageous, however, there is a method to make it efficient for a few modular multiplications by eliminating the need for these transformations [38]. The advantage of the Montgomery multiplication becomes much more apparent in applications requiring multiplication-intensive calculations, e.g., modular exponentiation or elliptic curve point operations. In order to exploit this advantage, all arithmetic operations are performed in the Montgomery domain, including the inversion operation [18, 47]. Furthermore, it is also possible to design cryptosystems in which all calculations are performed in the Montgomery domain eliminating the transformation operations permanently.

Below, we give bitwise Montgomery multiplication algorithm for obtaining $C := ABR^{-1} \pmod{p}$, where $A = (a_{m-1}, \ldots, a_1, a_0)$, $B = (b_{m-1}, \ldots, b_1, b_0)$, and $C = (c_{m-1}, \ldots, c_1, c_0)$.

Input:	$A, B \in GF(p)$ and $m = \lceil \log_2 p \rceil$
Output:	$C \in GF(p)$
Step 1:	$C := 0$
Step 2:	for $i = 0$ to $m - 1$
Step 3:	$C := C + a_i B$
Step 4:	$C := C + c_0 p$
Step 5:	$C := C/2$
Step 6:	if $C \geq p$ then $C := C - p$
Step 7:	return C

In the case of $GF(2^m)$, the definitions and the algorithms are slightly different since we use polynomials of degree at most $m - 1$ with coefficients from the binary field $GF(2)$ to represent the field elements. Given two polynomials

$$\begin{aligned} A(x) &= a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_1 x + a_0 \\ B(x) &= b_{m-1}x^{m-1} + b_{m-2}x^{m-2} + \cdots + b_1 x + b_0 \ , \end{aligned}$$

and the irreducible monic degree-m polynomial

$$p(x) = x^m + p_{m-1}x^{m-1} + p_{m-2}x^{m-2} + \cdots + p_1 x + p_0$$

generating the field $GF(2^m)$, the Montgomery multiplication of $A(x)$ and $B(x)$ is defined as the field element $C(x)$ which is given as

$$C(x) = A(x) \cdot B(x) \cdot R(x)^{-m} \pmod{p(x)} . \tag{3.3}$$

We note that, as compared to Equation (3.1), $R(x) = x^m$ replaces $R = 2^m$. The representation of x^m in the computer is exactly the same as the representation of 2^m, i.e., a single 1 followed by 2^m zeros. Furthermore, the elements of $GF(p)$ and $GF(2^m)$ are represented using the same data structures. For example, the elements of $GF(7)$ for $p = 7$ and the elements of $GF(2^3)$ for $p(x) = x^3 + x + 1$ are represented in the computer as follows:

$$\begin{aligned} GF(7) &= \{000, 001, 010, 011, 100, 101, 110\} , \\ GF(2^3) &= \{000, 001, 010, 011, 100, 101, 110, 111\} . \end{aligned}$$

Only the arithmetic operations acting on the field elements differ. The Montgomery image of a polynomial $A(x)$ is given as $\bar{A}(x) = A(x) \cdot x^m \pmod{p(x)}$. Similarly, before performing Montgomery multiplication, the operands must be transformed into the Montgomery domain and the result must be transformed back. These transformations are accomplished using the precomputed variable $R^2(x) = x^{2m} \pmod{p(x)}$ as follows:

$$\begin{aligned} \bar{A}(x) &= \mathsf{MonMul}(A, R^2) = A(x) \cdot R^2(x) \cdot R^{-1}(x) = A(x) \cdot R(x) \pmod{p(x)} , \\ \bar{B}(x) &= \mathsf{MonMul}(B, R^2) = B(x) \cdot R^2(x) \cdot R^{-1}(x) = B(x) \cdot R(x) \pmod{p(x)} , \\ C(x) &= \mathsf{MonMul}(\bar{C}, 1) = C(x) \cdot R(x) \cdot R^{-1}(x) = C(x) \pmod{p(x)} . \end{aligned}$$

The bit-level Montgomery multiplication algorithm for the field $GF(2^m)$ is given below:

Input:	$A(x), B(x) \in GF(2^m)$, $p(x)$, and m
Output:	$C(x)$
Step 1:	$C(x) := 0$
Step 2:	for $i = 0$ to $m - 1$

Step 3: $C(x) := C(x) + a_i B(x)$

Step 4: $C(x) := C(x) + c_0 p(x)$

Step 5: $C(x) := C(x)/x$

Step 6: return $C(x)$

We note that the extra subtraction operation in Step 6 of the previous algorithm is not required in the case of $GF(2^m)$, as proven in [25]. Also, the addition operations are different. While addition in binary field is just bitwise mod 2 addition, the addition in $GF(p)$ requires carry propagation.

Our basic observation is that it is possible to design a unified Montgomery multiplier which can perform multiplication in both types of fields if an adder module, equipped with the property of performing addition with or without carry, is available. The design of an adder with this property is provided in the following sections.

The algorithms presented in this section require that the operations be performed using full precision arithmetic modules, thus, limiting the designs to a fixed degree. In order to design a scalable architecture, we need modules with the scalability property. The scalable algorithms are word-level algorithms, which we give in the following sections.

3.4.1 Multiple-Word Montgomery Multiplication Algorithm for $GF(p)$

The use of fixed precision words alleviates the broadcast problem in the circuit implementation. Furthermore, a word-oriented algorithm allows design of a scalable unit. For a modulus of m-bit precision, $e = \lceil m/w \rceil$ words (each of which is w bits) are required. The algorithm proposed in [55] scans the operand B (multiplicand) word-by-word, and the operand A (multiplier) bit-by-bit. The vectors involved in multiplication operations are expressed as

$$B = (B^{(e-1)}, \ldots, B^{(1)}, B^{(0)}) ,$$

$$A = (a_{m-1}, \ldots, a_1, a_0) ,$$

$$p = (p^{(e-1)}, \ldots, p^{(1)}, p^{(0)}) ,$$

where the words are marked with superscripts and the bits are marked with subscripts. For example, the ith bit of the kth word of B is represented as $B_i^{(k)}$. A particular range of bits in a vector B from position i to j where $j > i$ is represented as $B_{j..i}$. Finally, 0^m represents an all-zero vector of m bits. The algorithm is given below:

Input: $A, B \in GF(p)$ and p

Output: $C \in GF(p)$

Step 1: $(TC, TS) := (0^m, 0^m)$

Step 2: $(Carry0, Carry1) := (0, 0)$

Step 3: for $i = 0$ to $m - 1$

Step 4: $\quad (TC^{(0)}, TS^{(0)}) := a_i \cdot B^{(0)} + TC^{(0)} + TS^{(0)}$

Step 5: $\quad Carry0 := TC_{w-1}^{(0)}$

Step 6: $\quad TC^{(0)} := (TC_{w-2..0}^{(0)} | 0)$

Step 7: $\quad parity := TS_0^0$

Step 8: $\quad (TC^{(0)}, TS^{(0)}) := parity \cdot p^{(0)} + TC^{(0)} + TS^{(0)}$

Step 9: $\quad TS_{w-2..0}^{(0)} := TS_{w-1..1}^{(0)}$

Step 10: $\quad$ for $j = 1$ to $e - 1$

Step 11: $\quad\quad (TC^{(j)}, TS^{(j)}) := a_i \cdot B^{(j)} + TC^{(j)} + TS^{(j)}$

Step 12: $\quad\quad Carry1 := TC_{w-1}^{(j)}$

Step 13: $\quad\quad TC_{w-1..1}^{(j)} := TC_{w-2..0}^{(j)}$

Step 14: $\quad\quad TC_0^{(j)} := Carry0$

Step 15: $\quad\quad Carry0 := Carry1$

Step 16: $\quad\quad (TC^{(j)}, TS^{(j)}) := parity \cdot p^{(j)} + TC^{(j)} + TS^{(j)}$

Step 17: $\quad\quad TS_{w-1}^{(j-1)} := TS_0^{(j)}$

Step 18: $\quad\quad TS_{w-2..0}^{(j)} := TS_{w-1..1}^{(j)}$

Step 19: $\quad$ end for

Step 20: $\quad TS_{w-1}^{(e-1)} := 0$

Step 21: end for

Step 22: $C := TC + TS$

Step 23: if $C > p$ then

Step 24: $C := C - p$

Step 25: return C

As suggested in [55], we use the Carry-Save form in order to represent the intermediate results in the algorithm. The result of an addition is stored in two variables $(TC^{(j)}, TS^{(j)})$, thus, they can grow as large as $2^{m+1} + 2^m - 3$ which is exactly equal to the result of the addition of three numbers at the right hand side of equations in Steps 4, 8, 11, and 16. Recall that $TC^{(j)}$ and $TS^{(j)}$ are m-bit numbers, but $TC^{(j)}$ must be seen as a number multiplied by 2 since it represents the carry vector in the Carry-Save notation. At the end of Step 21, we obtain the result in the Carry-Save form which needs an extra addition to get the final result in the nonredundant form. If the final result is greater than the modulus p, one subtraction operation must be performed as shown in Step 24.

3.4.2 Multiple-Word Montgomery Multiplication Algorithm for $GF(2^m)$

The Montgomery multiplication algorithm for $GF(2^m)$ is given below. Since there is no carry computation in $GF(2^m)$ arithmetic, the intermediate addition operations are replaced by bitwise XOR operations, which are represented below using the symbol $\oplus$.

Input: $A, B \in GF(2^m)$ and $p(x)$

Output: $C \in GF(2^m)$

Step 1: $TS := 0^m$

Step 2: for $i = 0$ to m

Step 3: $\quad TS^{(0)} := a_i B^{(0)} \oplus TS^{(0)}$

Step 4: $\quad parity := TS_0^{(0)}$

Step 5: $\quad TS^{(0)} := parity \cdot p^{(0)} \oplus TS^{(0)}$

Step 6: $\quad TS^{(0)}_{w-2..0} := TS^{(0)}_{w-1..1}$

Step 7: $\quad$ for $j = 1$ to $e - 1$

Step 8: $\quad\quad TS^{(j)} := a_i B^{(j)} \oplus TS^{(j)}$

Step 9: $\quad\quad TS^{(j)} := parity \cdot p^{(j)} \oplus TS^{(j)}$

Step 10: $TS_{w-1}^{(j-1)} := TS_0^{(j)}$

Step 11: $TS_{w-2..0}^{(j)} := TS_{w-1..1}^{(j)}$

Step 12: end for

Step 13: $TS_{w-1}^{(e-1)} := 0$

Step 14: end for

Step 15: $C := TS$

Step 16: return C

Notice that in the outer loop the index i runs from 0 to m. Since $(m+1)$ bits are required to represent irreducible polynomial of $GF(2^m)$, we prefer to allocate $(m+1)$ bits to express the field elements. We can also modify the algorithm for $GF(p)$ accordingly for sake of uniformity. Therefore, the formula for the number of words to represent a field element for both cases is given as $e = \lceil (m+1)/w \rceil$ where w is the selected wordsize.

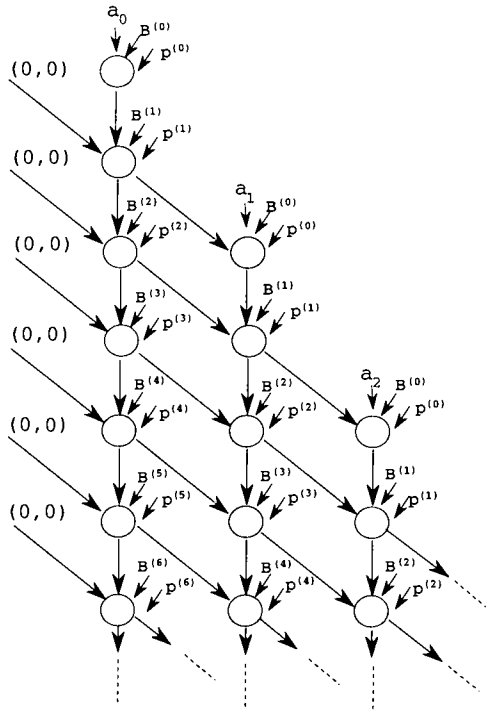
3.5 Concurrency in Montgomery Multiplication

In this section, we analyze the concurrency in Montgomery multiplication algorithms as given in the subsections §3.4.1 and §3.4.2. In order to accomplish this task, we need to determine the inherent data dependencies in the algorithm and describe a scheme to allow the Montgomery multiplication to be computed on an array of processing units organized in a pipeline.

We prefer to accomplish concurrent computation of the Montgomery multiplication by exploiting the parallelism among the instructions across the different iterations of i-loop of the algorithms, as proposed in [55]. We scan the multiplier one bit at a time, and after the first words of the intermediate variables (TC, TS) are fully determined, which takes two clock cycles, the computation for the second bit of A can start. In other words, after the inner loop finishes the execution for $j = 0$ and $j = 1$ in ith iteration of the outer loop, the $(i+1)th$ iteration of outer

loop starts its execution immediately. The dependency graph shown in Figure 3.1 illustrates these computations.

Figure 3.1. The dependency graph of the MonMul algorithm.

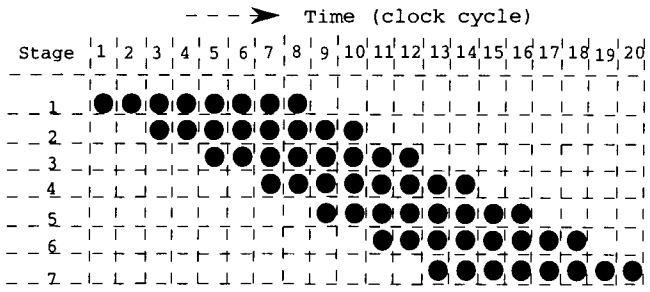


Each circle in the graph represents an elementary computation performed in each iteration of the j-loop. We observe from this graph that these computations are very suitable for pipelining. Each column in the graph represents operations that can be performed by separate processing units (PU) organized as a pipeline. Each PU takes only one bit from multiplier A and operates on each word of multiplicand, B, each cycle. Starting from the second clock cycle, a PU generates one word of partial sum $T = (TC, TS)$ in the Carry-Save form at each cycle, and communicates it to the next PU which adds its contribution to the partial sum, when its turn comes. After $e + 1$ clock cycles, the PU finishes its portion of work, and becomes

available for further computation. In case there is no available PU and there is work to do, the pipeline must stall and wait for the working PUs to finish their jobs. Since the PU at the end of the pipeline has no way of communicating its result to another PU, we need to provide extra buffers for them. In the worst case, which happens when there is only one PU, there must be $2e$ extra buffers of w length to hold these partial sum words. In the last clock cycle of each column, the The PU responsible for this column must receive $p^{(e)} = B^{(e)} = 0$. Elementary computations represented by circles in Figure 3.1 are performed on the same hardware module. Local control module in the PU must be able to extract $TS_0^{(0)}$ and keep this value for the entire operand scanning. Each PU, in other words, has to obtain this value and use it to decide whether to add the modulus p to the partial sum. This value is determined in the first clock cycle of the each stage.

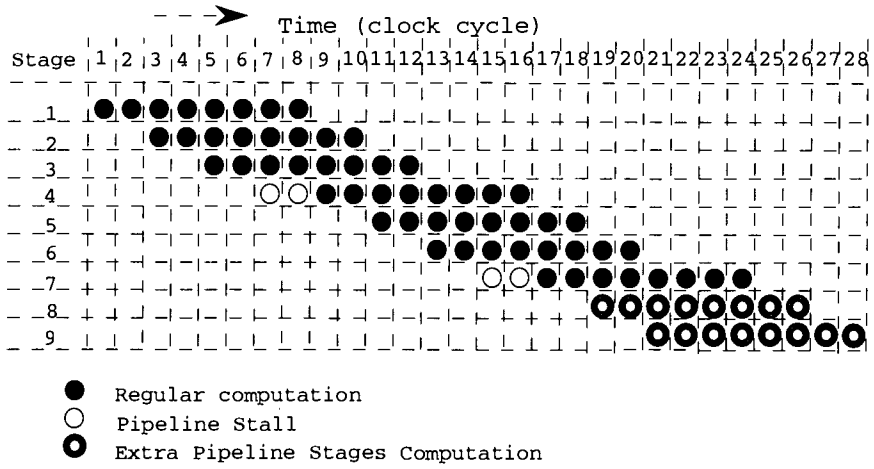
An example of the computation for 7-bit operands is shown in Figure 3.2 for the word size $w = 1$ provided that there are sufficient number of PUs preventing the pipeline to stall. Note that there is a delay of 2 clock cycles between the stage for x_i and the stage for x_{i+1}. The total execution time for the computation takes 20 clock cycles in this example.

Figure 3.2. An example of pipeline computation for 7-bit operands, where $w = 1$.



If there are at least $\lceil (e+1)/2 \rceil$ PUs in the pipeline organization the pipeline stalls do not take place. For the example in Figure 3.2, less than $\lceil 8/2 \rceil = 4$ PUs cause the pipeline to stall. Figure 3.3 shows what happens if there are only three PUs available for the same example.

Figure 3.3. An example of pipeline computation for 7-bit operands, illustrating the situation of pipeline stalls, where $w = 1$.



At the clock cycles 7 and 15, the pipeline cannot engage a PU, and thus, it must stall for 2 extra cycles. At the 9th and 17th cycles, the first PU becomes available and computation proceeds. We need a buffer of 4-bit length to store the partial sum bits during the stall. Because the 8 is not a multiple of 3, the last two pipeline stages perform extra computations. Since it is a pipeline organization, it is not possible to stop the computations at any time. In [55], these extra cycles are treated as waste cycles. However, it is possible to perform useful computation without complicating the circuit. Recall that $C = A \cdot B \cdot 2^{-m} \pmod{p}$ where m is the number of bits in the modulus p. If we continue the computations in these extra pipeline cycles, we calculate $C = A \cdot B \cdot 2^{-n} \pmod{p}$ where $n > m$ is the smallest

integer multiple of the number of PUs in the pipeline organization. It is always easy to rearrange the Montgomery settings according to this new Montgomery exponent, namely $R = 2^n$, or $R = x^n$ for the field $GF(2^m)$ case.

The total computation time, CC (clock cycles), is slightly different from the one in [55] and is given as

$$CC = \begin{cases} (\lceil \frac{m+1}{k} \rceil - 1)2k + e + 1 + 2(k-1) & \text{if } (e+1) < 2k \ , \\ (\lceil \frac{m+1}{k} \rceil)(e+1) + 2(k-1) & \text{otherwise} \ , \end{cases}$$

where k is the number of PUs in the pipeline. Notice that the first line of the formula gives the execution time in clock cycles when there are sufficiently many PUs while the second line corresponds to the case when there are stalls in the pipeline. At certain clock cycles some of the PUs become idle, and this affects the utilization of the unit, which can be formulated as

$$U = \frac{\text{Total number of clock cycles per bit of } A \times m}{\text{Total number of clock cycles } \times k} = \frac{(e+1) \cdot m}{CC \cdot k} \ .$$

Figure 3.4. The performance of multiple units with $w = 32$.

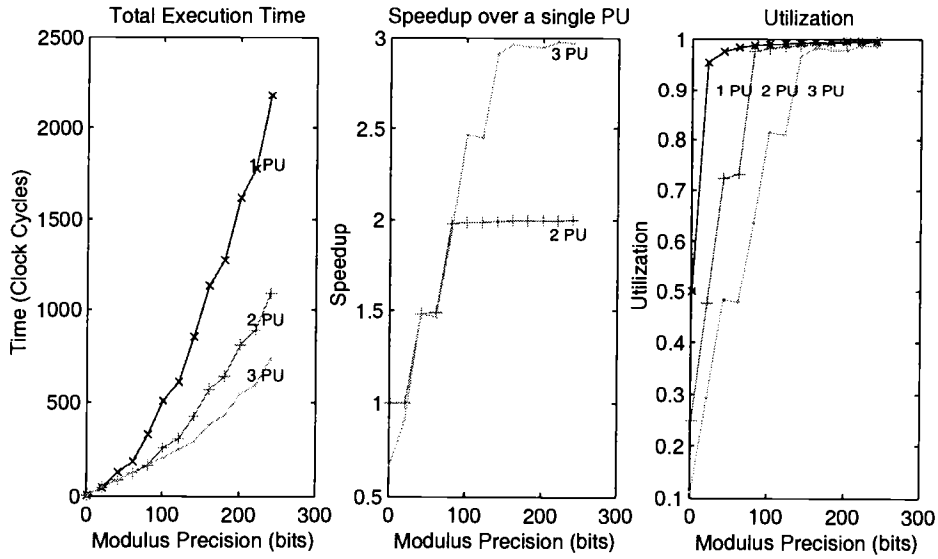
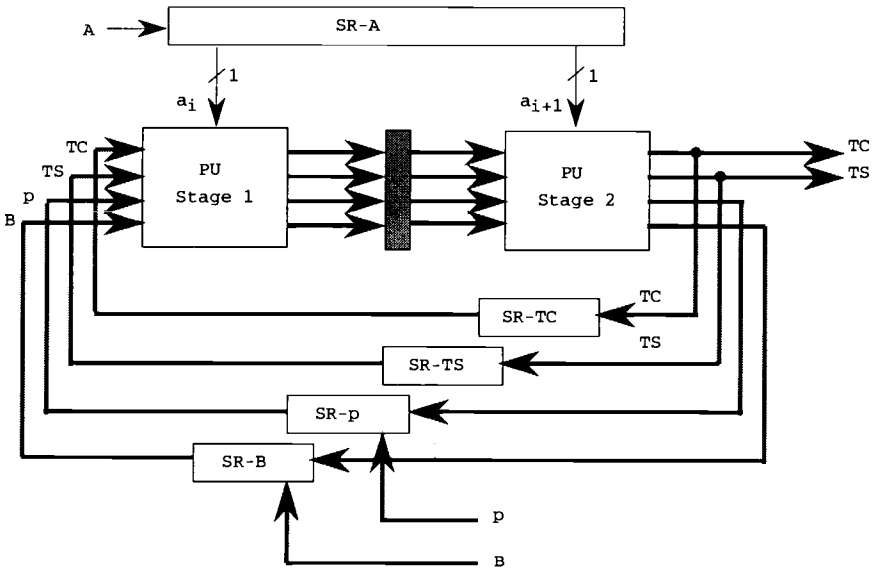


Figure 3.4 shows (from left to right) the total execution time CC, the speedup introduced by use of more units over a single unit, and the hardware utilization U for a range of precision. We prefer to select the wordsize as $w = 32$ in order to provide a realistic example, considering that most multi-purpose microprocessors have 32-bit datapaths.

3.6 Scalable Architecture

Figure 3.5. Pipeline organization with 2 PUs.



An example of pipeline organization with 2 PUs is shown in Figure 3.5. An important aspect of this organization is the register file design. The bits of multiplier a_i are given serially to the PUs, and are not used again in later stages and can be discarded immediately. Therefore, a simple shift register would be sufficient for the multiplier. The registers for the modulus p and multiplicand B can also be shift registers. When there is no pipeline stall, the latches between PUs forward the

modulus and multiplicand to next PU in the pipeline. However, if pipeline stalls occur, the modulus and multiplicand words generated at the end of the pipeline enter the $SR-p$ and $SR-B$ registers. The length of these shift registers are of crucial importance and determined by the number of pipeline stages (k) and the number of words (e) in the modulus. By considering that $SR-p$ and $SR-B$ values require one extra register to store the all-zero word needed for the last clock cycle in every stage (recall that $p^{(e)} = B^{(e)} = 0$) the length of these registers can be given as

$$L_1 = \begin{cases} e+1-2\cdot(k-1) & \text{if } (e+1) < 2k\ , \\ 1 & \text{otherwise.} \end{cases} \tag{3.4}$$

The width of the shift registers is equal to w, the wordsize. Once the partial sum (TC, TS) is generated, it is transmitted to the next stage without any delay. However, we need two shift registers, $SR-TC$ and $SR-TS$, to hold the partial sums from the last stage until the job in the first stage is completed. The length (L_2) of the registers TC and TS is equal to L_1.

We observe that only at most one word of each operand is used in every clock cycle. This makes different design options possible. Since we intend to design a fully scalable architecture, we need to avoid restrictions on the operand size or deterioration of the performance. Also we assume that no prior knowledge is available about the prospective range of the operand precision. Since the length of the shift registers can vary with the precision, designing full-precision registers within the multiplier might not be a good idea. Instead, one can limit the length of these registers within the chip and use memory for the excessive words. If this method is adopted, the length of the registers no longer would depend on the precision and/or the number of stages. The words needed earlier are brought from memory to the registers first, and the successive groups of words are transferred during the computation. If the memory transfer rate is not sufficient, however, pipeline might stall.

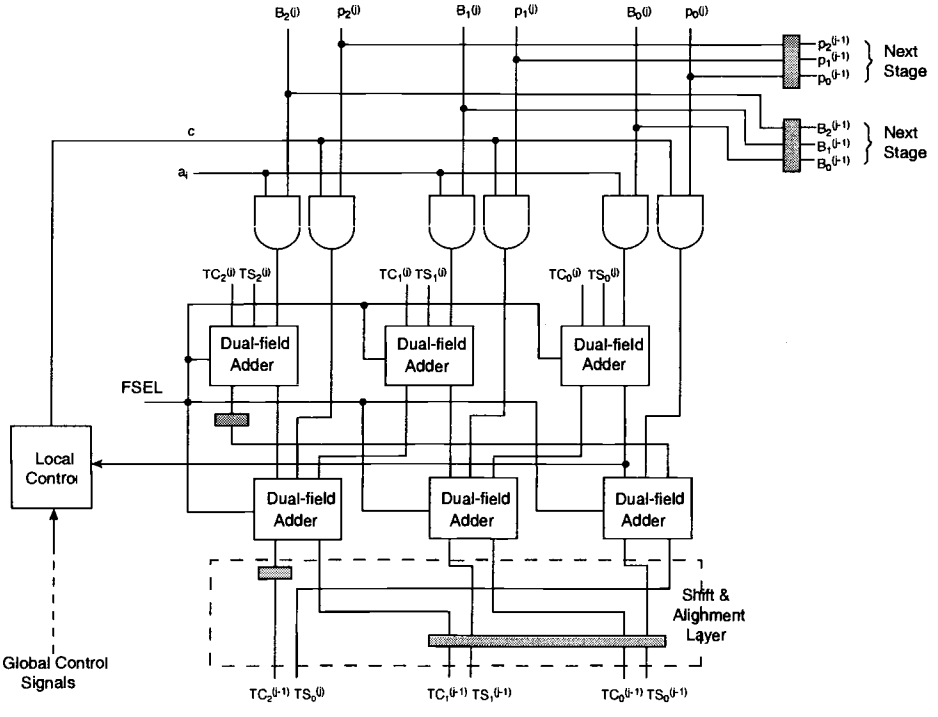
The registers for TC, TS, B, and p must have loading capability which can complicate the local control circuit by introducing several multiplexers (MUX). The

delay imposed by these MUXes will not create a critical path in the final circuit. The global control block was not mentioned since its function can be inferred from the dependency graph and the algorithms.

3.6.1 Processing Unit

The processing unit (PU) consists of two layers of adder blocks, which we call *dual-field adders*. A dual-field adder is basically a full adder which is capable of performing addition both with carry and without carry. Addition with carry corresponds to the addition operation in the field $GF(p)$ while addition without carry corresponds to the addition operation in the field $GF(2^m)$. We give the details about the dual-field adder in the next subsection. The block diagram of a processing unit (PU) for $w = 3$ is shown in Figure 3.6.

Figure 3.6. Processing Unit (PU) with $w = 3$.



The unit receives the inputs from the previous stage and/or from the registers $SR-A$, $SR-B$ and $SR-p$, and computes the partial sum words. It delays p and B for the first cycle, then, it transmits them to the next stage along with the first partial sum word (which is ready at the second clock cycle) if there is an available PU. The data path for partial sum $T=(TC,TS)$ (which is expressed in the redundant Carry-Save form) is $2w$ bits long while it is w bits long for p and B and 1 bit long for a_i. At the first cycle, the decision to add the modulus to the partial sum is determined, and this information is kept during the following e clock cycles. The computations in a PU for $e=5$ are illustrated in Table 3.1 for both types of fields $GF(p)$ and $GF(2^m)$.

Table 3.1. Inputs and outputs of the ith pipeline stage with $w=3$ and $e=5$ for both types of fields $GF(p)$ (top) and $GF(2^m)$ (bottom).

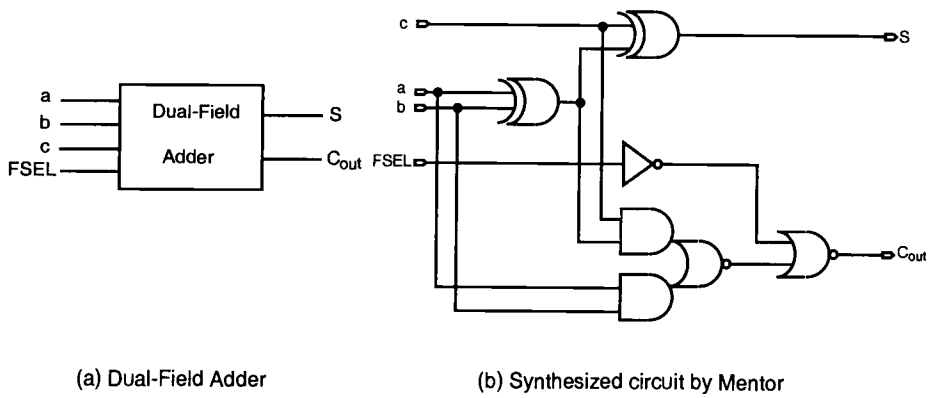
Cycle No	Inputs	Outputs
1	$TC^{(0)}, TS^{(0)}, a_i, B^{(0)}, P^{(0)}$	$(0, TS_0^{(0)})$; (0, 0); (0, 0)
2	$TC^{(1)}, TS^{(1)}, a_i, B^{(1)}, P^{(1)}$	$(TC_2^{(0)}, TS_0^{(1)})$; $(TS_2^{(0)}, TC_1^{(0)})$; $(TS_1^{(0)}, TC_0^{(0)})$
3	$TC^{(2)}, TS^{(2)}, a_i, B^{(2)}, P^{(2)}$	$(TC_2^{(1)}, TS_0^{(2)})$; $(TS_2^{(1)}, TC_1^{(1)})$; $(TS_1^{(1)}, TC_0^{(1)})$
4	$TC^{(3)}, TS^{(3)}, a_i, B^{(3)}, P^{(3)}$	$(TC_2^{(2)}, TS_0^{(3)})$; $(TS_2^{(2)}, TC_1^{(2)})$; $(TS_1^{(2)}, TC_0^{(2)})$
5	$TC^{(4)}, TS^{(4)}, a_i, B^{(4)}, P^{(4)}$	$(TC_2^{(3)}, TS_0^{(4)})$; $(TS_2^{(3)}, TC_1^{(3)})$; $(TS_1^{(3)}, TC_0^{(3)})$
6	0, 0, 0, 0, 0	$(TC_2^{(4)}, 0)$; $(TS_2^{(4)}, TC_1^{(4)})$; $(TS_1^{(4)}, TC_0^{(4)})$
1	$TC^{(0)}, TS^{(0)}, a_i, B^{(0)}, P^{(0)}$	$(0, TS_0^{(0)})$; (0, 0); (0, 0)
2	$TC^{(1)}, TS^{(1)}, a_i, B^{(1)}, P^{(1)}$	$(0, TS_0^{(1)})$; $(TS_2^{(0)}, 0)$; $(TS_1^{(0)}, 0)$
3	$TC^{(2)}, TS^{(2)}, a_i, B^{(2)}, P^{(2)}$	$(0, TS_0^{(2)})$; $(TS_2^{(1)}, 0)$; $(TS_1^{(1)}, 0)$
4	$TC^{(3)}, TS^{(3)}, a_i, B^{(3)}, P^{(3)}$	$(0, TS_0^{(3)})$; $(TS_2^{(2)}, 0)$; $(TS_1^{(2)}, 0)$
5	$TC^{(4)}, TS^{(4)}, a_i, B^{(4)}, P^{(4)}$	$(0, TS_0^{(4)})$; $(TS_2^{(3)}, 0)$; $(TS_1^{(3)}, 0)$
6	0, 0, 0, 0, 0	(0, 0); $(TS_2^{(4)}, 0)$; $(TS_1^{(4)}, 0)$

Notice that partial sum words in $GF(2^m)$ case are also in the redundant Carry-Save form. However, one of the components of the Carry-Save representation is always zero and the actual value of the result is the modulo-2 sum of the two.

Since consecutive operations are all additions and the Carry-Save form is already aligned by the shift and alignment layer, this does not lead to any problem. We need to recall, however, that one extra addition is necessary at the end of the multiplication process. In the next section, we introduce a multi-purpose word adder/subtractor module which performs this final addition at the cost of an extra clock cycle.

3.6.2 Dual-Field Adder

Figure 3.7. The dual-field adder circuit.



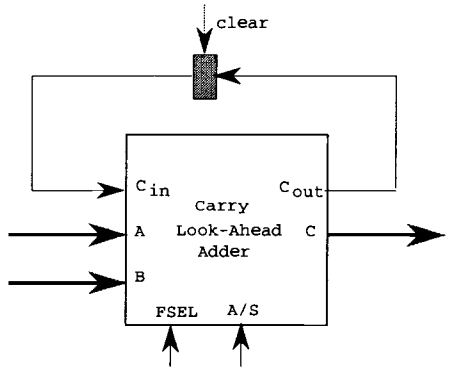
Dual-field adder (DFA) shown in Figure 3.7a, as mentioned before, is basically a full-adder equipped with the capability of doing bit addition both with and without carry. It has an input called $FSEL$ (field select) that enables this functionality. When $FSEL = 1$, the DFA performs the bit-wise addition with carry which enables the multiplier to do arithmetic in the field $GF(p)$. When $FSEL = 0$, on the other hand, the output C_{out} is forced to 0 regardless of the values of the inputs. The output S produces the result of bitwise modulo-2 addition of three input values. At most 2 of 3 input values of dual-field adder can have nonzero values while in the $GF(2^m)$ mode. An important aspect of designing the dual-field adder is not

to increase the critical path of the circuit which can have an effect on the clock speed which would be against our design goal. However, a small amount of extra area can be sacrificed. We show in the following section that this extra area is very insignificant. Figure 3.7b shows the actual circuit synthesized by Mentor Graphics tools using the $1.2\mu m$ CMOS technology.

In the circuit, the two XOR gates are dominant in terms of both area and propagation time. As in the standard full-adder circuit, the dual-field adder has two XOR gates connected serially. Thus, propagation time of the dual-field adder is not larger than that of full adder. Their areas differ slightly, but this does not cause a major change in the whole circuit.

3.7 Multi-purpose Word Adder/Subtractor

Figure 3.8. The word adder for field addition.

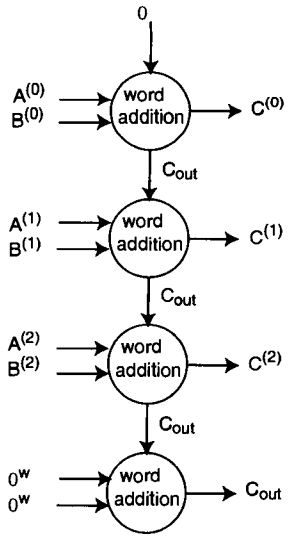


The proposed Montgomery multiplier generates results in the redundant Carry-Save form, hence we need to perform an extra addition operation at the end of the calculation to obtain the nonredundant form of the result. Therefore, a field adder circuit that operates in both $GF(p)$ and $GF(2^m)$ is necessary. A full-precision

adder would increase the critical path delay and the area, and would also be hard to scale. A word adder of the type given in Figure 3.8 would be suitable for our implementation since the multiplier generates only one word at each clock cycle in the last stage of pipeline, thus we need to perform one word addition at a time.

The word adder has two control inputs $FSEL$ and A/S, which enable to select the field ($GF(p)$ or $GF(2^k)$) and to choose between the addition and subtraction when in $GF(p)$ mode, respectively. The adder propagates the carry bit to the next word additions while working in $GF(p)$ mode (i.e., $FSEL = 1$). Thus, the carry from a word addition operation is delayed using a latch and fed back into the C_{in} input of the adder for the next word addition at the next clock cycle. In the $GF(2^m)$ mode, the module performs only bitwise modulo-2 addition of two input words and the A/S input is ineffective. An addition operation of two e-word long numbers takes $e+1$ clock cycles. The last cycle generates the carry and prepares the circuit for another operation by zeroing the output of latch. Figure 3.9 shows an example of addition operation with operands of 3 words.

Figure 3.9. An example of multiprecision addition operation with $e = 3$.



We added subtraction functionality in the field $GF(p)$ to the word adder because the result might be larger than the modulus, and hence one final subtraction operation is necessary as shown in Step 24 of the algorithm in §3.4.1. We do not need this reduction in the $GF(2^m)$ case. The final subtraction operation takes place only if the result is larger than the modulus. Thus, a comparison operation, which can also be performed utilizing the multi-purpose word adder/subtractor, is required. However, the control circuitry to perform this conditional subtraction might be complicated, therefore, it might be placed outside of the Montgomery multiplier unit.

Another reason to include a multi-purpose word adder unit in the multiplier circuit is the fact that the field addition operation is also needed in many cryptographic applications. For example, in elliptic curve cryptosystems, the field addition and multiplication operations are performed successively, hence having the multiplier and adder in the same hardware unit will decrease the communication overhead. A word adder that has these properties is synthesized using the Mentor Graphics tools and the time and space requirements are obtained, which are given in Table 3.2.

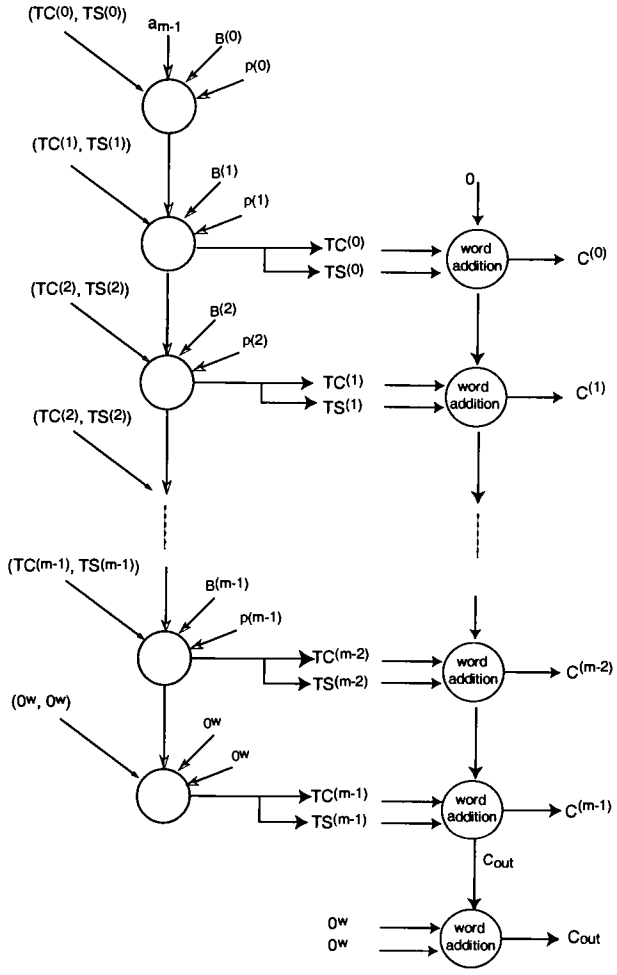
Table 3.2. Time and area costs of a multi-purpose word adder for $w = 16, 32, 64$.

bitsize	Propagation Time (ns)	Area (in NAND gates)
16	6.87	254
32	9.22	534
64	12.55	1128

Finally, Figure 3.10 illustrates what happens in last stage of the pipeline. A pair of redundant words $(TC_j^{(i)}, TS_j^{(i)})$ are generated each cycle for e clock cycles. The word adder can be used to add these pairs in order to obtain the result words $C^{(i)}$.

Note that only one extra cycle is needed to convert the result from the Carry-Save form to the nonredundant form.

Figure 3.10. Converting the result from the Carry-Save form to the nonredundant form in the last stage of the pipeline.



3.8 Design Considerations

In [55], an analysis of the area and time tradeoffs is given for the scalable multiplier. The architecture allows designs with different word lengths and different pipeline

organizations for varying values of operand precision. In addition, the area can be treated as a design constraint. Thus, one can adjust the design according to the given area, and choose appropriate values for the word length and the number of pipeline stages, in accordance. We give a similar analysis for the scalable and unified architecture. We are targeting two different classes of ranges for operand precision:

- *High precision range* which includes 512, 768 and 1024, is intended for applications requiring the exponentiation operation.
- *Moderate precision range* which includes 160, 192, 224, and 256, is typical for elliptic curve cryptography.

The propagation delay of the PU is independent of the wordsize w when w is relatively small, and thus all comparisons among different designs can be made under the assumption that the clock cycle is the same for all cases. The area consumed by the registers for the partial sum, the operands, and modulus is also the same for all designs, and we are not treating them as parts of the multiplier module.

The proposed scheme yields the worst performance for the case $w = m$ in the high precision range, since some extra cycles are introduced by the PU in order to allow word-serial computation, when compared to other full-precision conventional designs. On the other hand, using many pipeline stages with small wordsize values brings about no advantage after a certain point. Therefore, the performance evaluation reduces into finding an optimum organization for the circuit.

In order to determine the optimum selection for our organization, we obtain implementation results by synthesizing the circuit with Mentor Graphics tools using $1.2\mu m$ CMOS technology. The cell area for a given word size w is obtained as

$$A_{cell}(w) = 48.5w \tag{3.5}$$

units, and is slightly different from the one found in [55], where the multiplication factor in the formula is the area cost provided by the synthesis tool for a single bit

slice. Note that a 2-input NAND gate takes up 0.94 units of area. In the pipelined organization, the area of the inter-stage latches is important, which was measured as

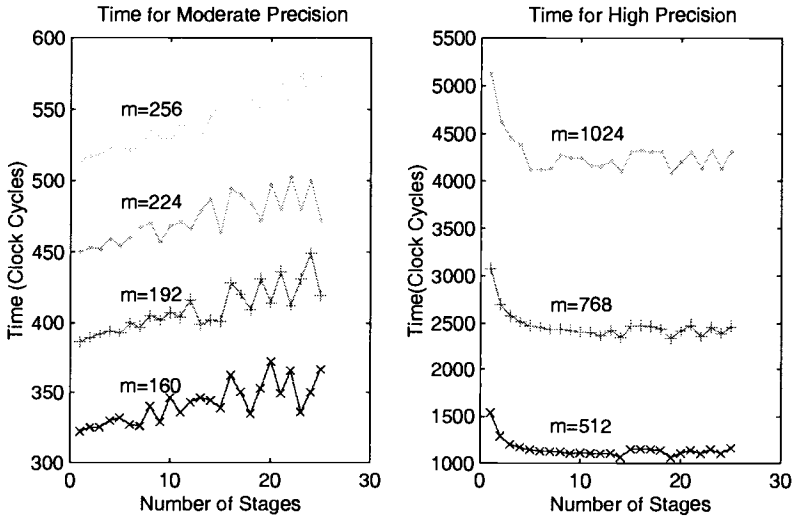
$$A_{latch}(w) = 8.32w \tag{3.6}$$

units. Thus, the area of a pipeline with k processing elements is given as

$$A_{pipe}(k, w) = (k-1)A_{latch}(w) + kA_{cell}(w) = 56,82kw - 8.32w \tag{3.7}$$

units. For a given area, we are able to evaluate different organizations and select the most suitable one for our application. The graphs given in Figure 3.11 allow to make such evaluations for a fixed area of 15,000 gates.

Figure 3.11. Time efficiency for different configurations with a fixed area of 15,000 gates.



For both moderate and high precision ranges, the number of stages between 5 and 10 are likely to give the best performance. For the high precision cases, fewer than 5 stages yields very poor performance since the fixed area becomes insufficient for large wordsizes and the performance degradation due to pipeline stalls becomes

a major problem. The small number of stages with very long word sizes seem to provide a reasonable performance in the moderate range, however, because of the incompatibility issues about using very long word sizes and inefficiency when the precision increases, using fewer than 5 stages is not advised. We avoid using many stages for two reasons:

- high utilization of the PUs will be possible only for very high precision, and
- the execution time may have undesirable oscillations.

The behavior mentioned in the latter category is the result of the facts that

- extra stages at the end of the computations, and
- there is not a good match between the number of words e and the number of stages k, causing a underutilization of stages in the pipeline.

From the synthesis tool we obtained a minimum clock cycle time of 11 nanoseconds, which allows to use a clock frequency of up to 90MHz with $1.2\mu m$ CMOS. Using the CMOS technology with smaller feature size, we can attain much faster clock speeds. It is very important to know how fast this hardware organization really is when comparing it to a software implementation. The answer to this would determine whether it is worth to design a hardware module. In general, it is difficult to compare hardware and software implementations. In order to obtain realistic comparisons, a processor which uses similar clock cycles and technology must be chosen. We selected an ARM microprocessor [11] with 80 MHz clock which has a very simple pipeline. We compare the $GF(p)$ multiplication timing on this processor against that of our hardware module. We use the same clock frequency 80 MHz for the module of the pipeline organization with $w = 32$ and $k = 7$ for the hardware module. On the other hand, the Montgomery multiplication algorithm is written in the ARM assembly language by using all known optimization techniques [24, 26]. Table 3.3 shows the multiplication timings and the speedup.

Table 3.3. The execution times of hardware and software implementations of the $GF(p)$ multiplication

precision	Hardware (μs) (80 MHz, $w = 32$, $k = 7$)	Software (μs) (on ARM with Assembly)	speedup
160	4.1	18.3	4.46
192	5.0	25.1	5.02
224	5.9	33.2	5.63
256	6.6	42.3	6.41
1024	61	570	9.34

3.9 Conclusion

Using the design methodology proposed in [55], we obtained a scalable field multiplier for $GF(p)$ and $GF(2^m)$ in unified hardware module. The methodology can also be used to design separate modules for $GF(p)$ and $GF(2^m)$ which are fast, scalable and area-efficient. The fundamental contribution of this research is to show that it is possible to design a dual-field arithmetic unit without compromising scalability, the time performance and area efficiency. We also presented a dual-field addition module which is suitable for the pipeline organization of the multiplier. The adder is scalable and capable of performing addition in both types of fields. Our analysis shows that a pipeline consisting of several stages is adequate and more efficient than a single unit processing very long words. Working with relatively short words diminishes data paths in the final circuit, reducing the required bandwidth.

The proposed multiplier was synthesized using the Mentor tools, and a circuit capable of working with clock frequencies up to 90 MHz is obtained. Except for the upper limit on the precision which is dictated only by the availability of memory to store the operands and internal results, the module is capable of performing infinite-precision Montgomery multiplication in $GF(2^m)$ and $GF(p)$.

Chapter 4
Efficient Methods for Composite Field Arithmetic

4.1 Introduction

Several algorithms for basic arithmetic operations in finite fields, suitable for hardware and software implementations have been recently developed [48, 57, 12, 43, 52, 58, 59]. The applications of these algorithms are found in error-correcting codes and public-key cryptography. In this chapter, we examine the existing methods and introduce new methods for software implementations of the arithmetic operations in the Galois field $GF(2^k)$. The proposed algorithms are suitable for obtaining high-speed implementations of the field operations on signal processors and microprocessors.

In this chapter, we consider a subset of the Galois fields $GF(2^k)$, the so-called *composite fields* where the exponent is a composite integer $k = nm$. It has been reported that efficient hardware and software implementations can be obtained for such fields [57, 12, 43, 42]. Two particular implementation methods are presented in [57, 12], where the field elements are represented as polynomials of length m with coefficients in the ground field $GF(2^n)$. The method in [57] carries out the field multiplication by first multiplying the input polynomials and reducing the resulting polynomial by a degree-m irreducible trinomial. On the other hand, the Karatsuba-Ofman algorithm is suggested for performing the multiplication operations in [12]. In both implementations, the logarithmic table lookup method is used for the ground field operations. The ground field is selected as $GF(2^{16})$ so that the coefficients of the elements in the composite representation would fit in a single computer word, which also makes the size of the logarithmic tables reasonable for general purpose computers. In order to decrease the complexity of the reduction

operation after the polynomial multiplication, m is selected so that the condition $\gcd(16, m) = 1$ holds. Unfortunately, this selection limits the possible number of fields (i.e., the values of k), where we can take the advantage of the composite representation. Furthermore, while the size of the lookup tables is still reasonable for $GF(2^{16})$, it would be more efficient to use smaller tables in order to take advantage of the first level cache in computers.

In this chapter, we improve the methods presented in [57, 12], and explore the implementation issues for more general cases. We have five main contributions:

1. We introduce new efficient table lookup methods using the values of n which are not an integer multiple of 8. For example, we can take n as 13, 14, and 15, which provide more combinations of n and m, and thus, more composite field implementations with varying efficiency values. This also necessitates a closer examination of the possible implementations in order to select the most efficient one.

2. We propose the use of the optimal normal bases (ONB1 and ONB2) in addition to the polynomial basis for the composite fields. Although the polynomial basis representation provides faster multiplication methods, the squaring operation in the ONB1 and ONB2 is significantly faster.

3. We propose the use of the new specialized algorithms [27, 54] for multiplication in ONB1 and ONB2, rather than the general purpose Massey-Omura multiplication algorithm [39]. The resulting methods are very efficient and provide comparable performance to the polynomial basis.

4. We propose a new and efficient method for inversion in the composite fields using the optimal normal basis. The new method is based on the extended Euclidean algorithm, and it is faster than the Itoh-Tsujii method [16].

5. We provide extensive timing results of our implementations for the composite fields in order to determine which n and m combinations would give better

performance. This provides alternative implementations of the fields which have the same size or the similar size.

4.2 Composite Fields

In this section, we summarize relevant properties of the composite fields. Let $GF(2^k)$ denote the binary extention field defined over the prime field $GF(2)$. If the elements of the set

$$B_1 = \{1, \alpha, \alpha^2, \ldots, \alpha^{k-1}\} \tag{4.1}$$

are linearly independent, then B_1 forms a polynomial basis for $GF(2^k)$. Thus, given an element $A \in GF(2^k)$, we can write

$$A = \sum_{i=0}^{k-1} a_i \, \alpha^i \, , \tag{4.2}$$

where $a_0, a_1, \ldots a_{k-1} \in GF(2)$ are the coefficients. Once the basis is chosen, the rules for the field operations (addition, multiplication, and inversion) can be derived.

There are various ways to represent the elements of $GF(2^k)$ depending on the choice of the basis or on the construction method of $GF(2^k)$. If k is the product of two integers as $k = nm$, then it is possible to derive a different representation method by defining $GF(2^k)$ over $GF(2^n)$. An extension field which is not defined over the prime field but one of its subfields is known as a composite field. It is denoted as $GF((2^n)^m)$ where $GF(2^n)$ is known as the ground field over which the composite field is defined. There is only one finite field of characteristic 2 for a given degree, and both the binary and composite fields refer to the same field although their representation methods are different. In order to represent the elements in the composite field $GF((2^n)^m)$, we can use the basis

$$B_2 = \{1, \beta, \beta^2, \ldots, \beta^{m-1}\} \, , \tag{4.3}$$

where β is the root of a degree m irreducible polynomial whose coefficients are in

the base field $GF(2^n)$. Thus, an element $A \in GF((2^n)^m)$ can be written as

$$A = \sum_{i=0}^{m-1} a'_i \cdot \beta^i , \tag{4.4}$$

where $a'_0, a'_1, \ldots, a'_{m-1} \in GF(2^n)$. Since the coefficients in the composite field representation are no longer in the prime field, we need to know how to calculate in the ground field $GF(2^n)$. The ground field operations are carried out using precalculated logarithmic lookup tables in composite field applications, and thus, the selection of the basis for the ground field is not important. However, in order to construct the logarithmic tables, we need to find a primitive element in $GF(2^n)$.

4.3 Arithmetic in the Ground Field

The logaritmic table lookup method for performing arithmetic in $GF(2^n)$ for small values of n is a well-known method [13, 57, 12]. A primitive element $g \in GF(2^n)$ is selected to serve as the generator of the field $GF(2^n)$, so that an element A in this field can be written as a power of g as $A = g^i$, where $0 \leq i \leq 2^n - 1$. Then, we compute the powers of the primitive element as g^i for $i = 0, 1, \ldots 2^n - 1$, and obtain 2^n pairs of the form (A, i).

We construct two tables sorting these pairs in two different ways: **the log table** sorted with respect to A and **the alog table** sorted with respect to i. For example, for $i = 5$ and $A = g^5$, we have $\log[A] = 5$ and $\text{alog}[5] = A$. These tables are then used for performing the field multiplication, the squaring, and the inversion operations. The tables are particularly very useful in software implementations. Given two elements $A, B \in GF(2^n))$, we perform the multiplication $C = AB$ as follows:

1. $i := \log[A]$
2. $j := \log[B]$
3. $k := i + j \pmod{2^n - 1}$
4. $C := \text{alog}[k]$

This is due to the fact that $C = AB = g^i g^j = g^{i+j \bmod 2^n-1}$. The ground field multiplication requires three memory access, a single modular addition operation with modulus $2^n - 1$. The squaring of an element A is slightly easier: only two memory access operations are required for computing $C = A^2$, as illustrated below:

1. $i := \log[A]$
2. $k := 2i \pmod{2^n - 1}$
3. $C := \text{alog}[k]$

Similarly, the inversion of an element A is computed using the property $C = A^{-1} = g^{-i} = g^{2^n-1-i}$, which requires two memory access operations:

1. $i := \log[A]$
2. $k := 2^n - 1 - i$
3. $C := \text{alog}[k]$

In order to speed the ground field operations, particularly the multiplication and the addition operations, we propose two improvements:

- The use of **the extended alog table**.

  The extended alog table eliminates the modular addition operation in the multiplication (Step 3) and the squaring (Step 2) operations. The extended alog table is of length $2^{n+1} - 1$ which is about the twice the length of the standard alog table. It contains the values (k, g^k) sorted with respect to the index k, where $k = 0, 1, 2, \ldots, 2^{n+1} - 2$. Since the values of i and j in Step 1 and 2 of the multiplications are in the range $[0, 2^n - 1]$, the range of $k = i + j$ is $[0, 2^{n+1} - 2]$. Therefore, there is no need for computing the modular addition operation, and the ground field multiplication operation is simplified as follows:

1. $i := \log[A]$
2. $j := \log[B]$
3. $k := i + j$
4. $C :=$ extended-alog$[k]$

Similarly, the squaring operation is given as

1. $i := \log[A]$
2. $k := 2i$
3. $C :=$ extended-alog$[k]$

The penalty paid for gaining the improved performance is the size of the extended alog table. It is twice the size of the standard alog table.

There is no particular reason to use the extended table for the inversion operation since $k = 2^n - 1 - i$ is still in the range $[0, 2^n - 1]$.

- The use of n values other than 8 or 16.

  The previous methods sugguest that we take n as 8 or 16 [13, 57]. We propose to use other values of n, particularly, $n = 13, 14, 15$ which are more useful for general purpose computer implementations. Since it is recommended to have relatively prime n and m, we obtain more composite fields with these choices of n. Furthermore, when we select n as 13, 14, or 15, we can also limit the size of the extended alog table to still fewer than 2^{16} words. Since the size is given as $2^{n+1} - 1$, the largest table becomes of $2^{15+1} - 1 = 2^{16} - 1$ words. We do not recommend the use of the extended alog table for $n = 16$ since in this case the length of the extended alog table becomes $2^{17} - 1 = 131,071$ words or $262,142$ bytes which may be considered excessive (may not fit most caches).

4.4 Polynomial Basis Representation of Composite Fields

The elements of $GF((2^n)^m)$ are treated as m-dimensional vectors over $GF(2^n)$ in the composite field representation. Since the coefficients in this representation are n-bit words, it is more advantageous for implementation on microprocessors, particularly when n is selected properly. In our implementation, we use the ground fields of degrees 13, 14, 15, and 16. Therefore, 16-bit computer words are sufficient to store the coefficients, however, we do not utilize a few bits in the most significant positions of the computer word. This is not a significant loss, but it can cause the number of words to represent composite field elements to increase, especially when $n = 13$. Since the arithmetic operations have to handle more words this can slow down the implementation. Using smaller lookup tables, on the other hand, will have better performance due to the localization of the memory access.

In the polynomial basis implementation, an irreducible polynomial of degree m with coefficients from $GF(2^n)$ is chosen to perform arithmetic operations in $GF((2^n)^m)$. An m-th degree polynomial which is irreducible over $GF(2)$ is also irreducible over $GF(2^n)$ if $\gcd(n, m) = 1$. Since we use both even and odd numbers between 13 and 16 for n such selections of n and m do not limit m to odd numbers as in [57]. We tabulate all possible composite field degrees between 160 and 512 for $n = 13, 14, 15, 16$ in Table 4.1. The rule for obtaining Table 4.1 is as follows:

- $\gcd(n, m) = 1$ and $n \in [13, 16]$ and $nm \in [160, 512]$.

These are the composite fields for which we can produce efficient implementations by selecting the aforementioned values of n. This gives much more composite fields than we can obtain by using $n = 16$ only.

The use of the composite field representation substantially speeds up the reduction operation following the polynomial multiplication operation. The reduction operation can be accelerated even further if an irreducible trinomial or pentanomial is used. It is established that for each integer $m \in [2, 10000]$ there exists either an irreducible trinomial or pentanomial [51]. In our implementation, we

performed arithmetic in the composite fields using the polynomial basis similar to [57] whenever the field polynomial is an irreducible trinomial. However, when there does not exist an irreducible trinomial for a particular degree of m, we used an irreducible pentanomial. It is observed that the performance is still good for pentanomials.

We give the timing results for the field operations in Table 4.2 for a subset of the composite fields enumerated in Table 4.1. As can be observed from Table 4.2, the advantage of using the values of n other than 16 is apparent. For example, while the multiplication for $(n, m) = (16, 15)$ takes 14.1 microseconds, it takes only 10.8 microseconds for $(n, m) = (15, 16)$. Since lookup tables for $n = 15$ are smaller than those for $n = 16$, the memory access times are shorter, hence the multiplication is faster.

4.5 Optimal Normal Basis Representation of Composite Fields

A normal basis for the binary field $GF(2^k)$ is given as

$$B = \{\beta, \beta^2, \beta^{2^2}, \ldots, \beta^{2^{k-1}}\} \tag{4.5}$$

with the property that the elements of B are linearly independent. There exists at least one normal basis for $GF(2^k)$ for every positive integer k. The normal basis representations have the computational advantage that the squaring of an element can be performend by a circular shift. On the other hand, the multiplication of two distinct elements requires a more complicated circuit, whose complexity is reduced only for a subset of normal bases, called the optimal normal bases [31].

There exists two types of the optimal normal bases which are historically named as the optimal normal basis of type 1 (ONB1) and the optimal normal basis of type 2 (ONB2). There are 117 and 319 m values in the range $m \in [2, 2001]$, such that the field $GF(2^m)$ has an optimal normal basis of type 1 and type 2, respectively [31]. the ONB2 is more abundant, thus, this representation is much more useful.

Table 4.1. Composite field degrees ($165 < k < 512$) using the polynomial basis.

n	m	nm	n	m	nm	n	m	nm	n	m	nm
13	14	182	14	13	182	15	11	165	16	11	176
	15	195		15	210		14	210		13	208
	16	208		17	238		16	240		15	240
	17	221		19	266		17	255		17	272
	18	234		23	322		19	285		19	304
	19	247		25	350		22	330		21	336
	20	260		27	378		23	345		23	368
	21	273		29	406		26	390		25	400
	22	286		31	434		28	420		27	432
	23	299		33	462		29	435		29	464
	24	312					31	465		31	496
	25	325					32	480			
	27	351					34	510			
	28	364									
	29	377									
	30	390									
	31	403									
	32	416									
	33	429									
	34	442									
	35	455									
	36	468									
	37	481									
	38	494									

Table 4.2. The timings in microseconds for the polynomial basis

n	m	$k = nm$	Squaring	Multiplication	Inversion
13	14	182	1.29	7.6	29
	15	195	1.28	8.6	32
	16	208	1.70	9.6	36
	18	234	1.56	12.1	43
	23	299	1.84	18.6	66
	28	364	2.23	24.4	93
	29	377	2.30	25.9	99
	30	390	2.38	27.6	104
	33	429	2.40	32.9	125
	35	455	2.49	36.4	137
	36	468	2.58	38.3	148
	38	494	2.97	43.1	164
14	13	182	1.53	7.4	28
	15	210	1.37	9.1	35
	19	266	2.02	14.2	53
	23	322	1.97	19.4	72
	27	378	2.73	25.7	96
	29	406	2.46	28.2	109
	33	476	2.66	35.6	137
15	11	165	1.20	5.8	23
	13	195	1.59	7.6	29
	14	210	1.44	8.4	33
	16	240	1.88	10.8	42
	19	285	2.08	14.6	55
	23	345	2.05	19.9	75
	26	390	2.75	24.4	100

Table 4.2: The timings in microseconds for the polynomial basis(cont.)

n	m	nm	Squaring	Multiplication	Inversion
15	28	420	2.49	27.2	107
	29	435	2.57	29.0	113
	31	465	2.34	33.2	127
	34	510	2.63	38.7	151
16	11	176	1.44	8.9	30
	13	208	1.79	11.0	40
	15	240	1.79	14.1	50
	23	368	2.60	31.0	97
	27	432	3.25	42.1	127
	29	464	2.90	46.5	144
	31	496	3.30	54.9	164

The composite field elements can be represented using optimal normal bases when there exists an ONB1 or ONB2 for $GF(2^m)$ in the setting $GF((2^n)^m)$. As long as $\gcd(n, m) = 1$, a linearly independent set which forms an optimal normal basis for a binary field $GF(2^m)$ also forms an optimal normal basis for the composite field $GF((2^n)^m)$. The element A in the composite field can be written as

$$A = \sum_{i=0}^{m-1} A_i \, \beta^i \, , \tag{4.6}$$

where $A_i \in GF(2^n)$. Since the degree-m normal polynomial is also a normal polynomial in $GF((2^n)^m)$, every algorithm for performing arithmetic in the binary field for the normal bases also works in the composite field without any modification.

In Table 4.3, we enumerate all possible composite fields expressed using the ONB1 and ONB2 for $160 < nm < 512$, where the ground field is taken as $GF(2^n)$ for $n = 13, 14, 15, 16$. As expected, we have more composite fields expressed in ONB2 than in ONB1 in this range.

Table 4.3. Composite field degrees ($160 < k < 512$) using the ONB1 and ONB2

ONB1			ONB2		
n	m	nm	n	m	nm
13	18	234	13	14	182
	28	364		18	234
	36	468		23	299
15	28	420		29	377
				30	390
				33	429
				35	455
			14	23	322
				29	406
				33	462
			15	11	165
				14	210
				23	345
				26	390
				29	435
			16	11	176
				23	368
				29	464

The rule for obtaining Table 4.3 is as follows:

- ONB1 for $GF((2^n)^m)$: An ONB1 exists for $GF(2^m)$ and $\gcd(n, m) = 1$ and $n \in [13, 16]$ and $nm \in [160, 512]$.
- ONB2 for $GF((2^n)^m)$: An ONB2 exists for $GF(2^m)$ and $\gcd(n, m) = 1$ and $n \in [13, 16]$ and $nm \in [160, 512]$.

In the following, we present the squaring, multiplication, and inversion algorithms for the optimal normal bases, which are used in obtaining the composite field implementations. The Massey-Omura [39] algorithm can be used for multiplication of elements represented using the ONB1 and ONB2, however, there are also specialized algorithms [27, 54]. We promote the use of these specialized algorithms for the multiplication operation in the composite fields.

4.5.1 Squaring in ONB1 and ONB2

The squaring in a normal basis is simply a bitwise circular shift of the binary vector. For the composite fields, the squaring is performed using a circular word shift after each word is squared in the ground field $GF(2^n)$ using the lookup tables. This squaring operation is significantly more efficient than the one in the polynomial basis because it does not require a modular reduction. Let $A \in GF((2^n)^m)$ be represented using an m-dimensional vector as

$$A = (A_0, A_1, \ldots, A_{m-2}, A_{m-1}) \ , \tag{4.7}$$

where $A_i \in GF(2^n)$ for $i = 0, 1, \ldots, m-1$. Using the property $\beta^{2^m} = \beta$, we obtain A^2 as follows:

$$\begin{aligned} A^2 &= (\sum_{i=0}^{m-1} A_i \beta^{2^i})^2 = \sum_{i=0}^{m-1} A_i^2 \beta^{2^{i+1}} \\ &= (A_{m-1}^2, A_0^2, A_1^2, ..., A_{m-3}^2, A_{m-2}^2) \ . \end{aligned}$$

4.5.2 Multiplication in ONB1

We use the algorithm proposed in [27] in our implementation. Here we give a brief description of this multiplication algorithm. An ONB1 is generated by an element $\beta \in GF((2^m)^n)$ of order $p = m + 1$. Since 2 is primitive modulo p, the set which is the basis for the ONB1 representation

$$N = (\beta, \beta^2, \beta^{2^2}, ..., \beta^{2^{m-1}}) \tag{4.8}$$

is equivalent to the set

$$M = (\beta, \beta^2, \beta^3, ..., \beta^{m-1}) \ . \tag{4.9}$$

The set M is called the shifted polynomial basis, and its field polynomial is an irreducible all-one-polynomial [27]. Furthermore, M is obtained from N by a permutation. Let the field element $A \in GF((2^n)^m)$ be expressed in ONB1 as

$$A = \sum_{i=0}^{m-1} A_i \beta^{2^i}. \tag{4.10}$$

We can also express A in the shifted polynomial basis as

$$\bar{A} = \sum_{i=0}^{m-1} \bar{A}_i \beta^{i+1} \ . \tag{4.11}$$

The conversion between them is established using the following permutation P:

$$\bar{A}_{(2^i-1) \pmod{m+1}} = A_i \quad \text{for} \quad \mathrm{i} = 0, 1, \ldots, \mathrm{m} - 1 \ . \tag{4.12}$$

The multiplication in the ONB1 reduces to the polynomial multiplication taking advantage of the arithmetic with an irreducible all-one-polynomial [27].

1. Obtain the shifted polynomial representation of A and B using permutation P.

2. Perform the polynomial multiplication and obtain $\bar{C}$.

3. Apply inverse permutation P^{-1} to $\bar{C}$ and obtain the result in the ONB1.

Since the permutation gives shifted polynomial representation of A and B, we need to perform an extra correction in Step 2 of the algorithm. Let $A, B \in GF((2^n)^m)$ be represented in the shifted polynomial basis as $A = (\bar{A}_0, \bar{A}_1, \ldots, \bar{A}_{m-1})$ and $B = (\bar{B}_0, \bar{B}_1, \ldots, \bar{B}_{m-1})$. After the multiplication operation, the result is obtained as $F = \bar{A}\bar{B}/\beta^2$, and represented in polynomial base as follows:

$$F = F_0 + F_1\beta + F_2\beta^2 + \ldots + F_{m-1}\beta^{m-1} .$$

In order to obtain the correct result, we first multiply F by β^2 (i.e., we shift F two words to the left), and obtain

$$E = F_0\beta^2 + F_1\beta^3 + \ldots + F_{m-1}\beta^{m+1} .$$

However, the result is still not in the shifted polynomial basis since the weight of the term F_{m-1} is β^{m+1}, which needs to be reduced using the relation

$$\beta^{m+1} = \beta + \beta^2 + \ldots + \beta^m .$$

Therefore, after the correction operation, we obtain the shifted polynomial representation of the result as

$$C = F_{m-1}\beta + (F_{m-1} + F_0)\beta + \ldots + (F_{m-1} + F_{m-2})\beta^m .$$

We then need to apply the inverse permutation P^{-1} to C, and obtain the final result expressed in ONB1.

4.5.3 Multiplication in ONB2

We used the multiplication algorithm proposed in [54] in this case. Since this algorithm is not published, we will provide a description. It is somewhat similar to the algorithm described in the previous section. It also requires a basis conversion from the ONB2 to a new type of basis. However, the new basis is not a polynomial basis, and the multiplication in this new basis is more complicated. An ONB2 for the field $GF((2^n)^m)$ is constructed using the normal element $\beta = \gamma + \gamma^{-1}$ where

γ is a primitive $(2m+1)$th root of unity, i.e. $\gamma^{2m+1} = 1$ and $\gamma^i \neq 1$ for any $1 \leq i \leq 2m+1$. It turns out that an ONB2 can be constructed if $p = 2m+1$ is prime and also if either of the following two conditions holds:

- 2 is primitive modulo p
- $p \equiv 3 \pmod 4$ and 2 generates the quadratic residues modulo p [31].

An element $A \in GF((2^n)^m)$ is represented using the ONB2

$$N = \{\beta, \beta^2, \ldots, \beta^{2^{m-1}}\} = \{\gamma + \gamma^{-1}, \gamma^2 + \gamma^{-2}, \gamma^{2^2} + \gamma^{-2^2}, \ldots \gamma^{2^{m-1}} + \gamma^{-2^{m-1}}\}$$

as follows:

$$A = A_0\beta + A_1\beta^2 + \ldots + A_{m-1}\beta^{2^{m-1}} .$$

A basis element can be written as $\beta^{2^i} = \gamma^j + \gamma^{-j}$ for $j \in [1, 2m]$ following the fact that 2 is primitive modulo p. The set

$$M = \{\gamma + \gamma^{-1}, \gamma^2 + \gamma^{-2}, \gamma^3 + \gamma^3, \ldots \gamma^m + \gamma^m\}$$

is a permutation of the ONB2 basis N, hence forms another basis for $GF((2^n)^m)$. Then $A \in GF((2^n)^m)$ is expressed in the new basis M as

$$A = \bar{A}_0\beta_1 + \bar{A}_1\beta_2 + \ldots + \bar{A}_{m-1}\beta_m .$$

where $\beta_i = \gamma^i + \gamma^{-i}$. The conversion from one representation to the other involves only a permutation, which can be given in terms of coefficients $\bar{A}_j = A_i$ as

$$j = \begin{cases} k & \text{if } k \in [1, m] \\ (2m+1) - k & \text{if } k \in [m+1, 2m] \end{cases} ,$$

where $k = 2^{i-1} \pmod p$ for $i = 1, 2, \ldots, m$. The multiplication in basis M is performed using the formulae which were derived in [54]. Let $A, B \in GF((2^n)^m)$ are represented in M as

$$A = \sum_{i=1}^{m} \bar{A}_i\beta_i \quad \text{and} \quad B = \sum_{i=1}^{m} \bar{B}_i\beta_i .$$

Then, the product $C = AB$ can be calculated using $C = C_1 + D_1 + D_2$, where

$$
\begin{aligned}
C_1 &= \sum_{\substack{1 \le i,j \le m \\ i \ne j}} \bar{A}_i \bar{B}_j \beta_{i-j} \\
D_1 &= \sum_{i=1}^{m} \sum_{j=1}^{m-i} \bar{A}_i \bar{B}_j \beta_{i+j} \\
D_2 &= \sum_{i=1}^{m} \sum_{j=m-i+1}^{m} \bar{A}_i \bar{B}_j \beta_{i+j}
\end{aligned}
$$

Then, the ONB2 representation of the result is obtained using the inverse permutation.

4.5.4 Inversion in ONB1 and ONB2

The Itoh-Tsujii algorithm [16] for inversion in the binary fields using the normal bases is also suggested for the composite fields using the polynomial basis in [12]. The algorithm reduces the inversion problem in the composite field to the inversion in the ground field $GF(2^n)$. However, it requires several field multiplications, and the number of these multiplications increases as m gets larger. On the other hand, the inversion algorithm for the polynomial basis described in [57] is very efficient, and it is possible to apply the same algorithm to calculate the inversion of the composite field elements expressed in a normal basis. In order to modify this algorithm for composite normal basis, an element given in the normal basis is transformed to the polynomial basis using the field polynomial of the normal basis. Although the field polynomial of the normal basis may have many non-zero terms, this is not a disadvantage since using a field polynomial which is not a trinomial or pentanomial does not slow down the inversion operation. The inversion operation is performed on the transformed element, and finally the result is mapped back into the normal basis. Although transformation and inverse transformation operations seem to complicate the calculation, our experimental results show that even for $m = 11$, which requires only four multiplications, we obtain better results with the latter algorithm.

The inversion algorithm in polynomial basis is based on the extended Euclidean algorithm, which can be given as follows:

- **Input:** $A \in GF((2^n)^m)$ and P (the irreducible field polynomial)
- **Output:** $B \in GF((2^n)^m)$ such that $AB = 1 \pmod{P}$

1. Initialize polynomials $B := 1$, $C := 0$, $F := A$, and $G := P$
2. if $F = 1$ then return $B \cdot F^{-1}$
3. if $\deg(F) < \deg(G)$ then exchange F & G and exchange B & C
4. $\delta := \deg(F) - \deg(G)$
5. $\alpha = F_{\deg(F)} \cdot G_{\deg(G)}$
6. $F := F + \alpha x^{\delta} G$ and $B := B + \alpha x^{\delta} C$
7. Go to Step 2

We implemented the Itoh-Tsujii and the extended Euclidean algorithms in the C language, and obtained some timing results using the Microsoft Visual C++ 5.0 on a 450-MHz Pentium II based PC running Windows NT 4.0. The timing values given in Table 4.4 are in microseconds. As can be observed from Table 4.4, the extended Euclidean algorithm is much faster when the subfield degree is large since the length of F decreases in each iteration.

In order to apply the polynomial inversion algorithm for optimal normal bases, the elements represented in the optimal normal basis need to be converted to the polynomial basis. The polynomial basis constructed using the field polynomial of the normal basis provides considerable advantage for this conversion because only XOR and assignment operations are required during the conversion. The field polynomials for the ONB1 bases are irreducible all-one-polynomials. On the other

Table 4.4. The inversion timings in microseconds

n	m	nm	Itoh-Tsujii	Extended Euclid
13	14	182	62	29
	18	234	108	45
15	11	165	33	22
	14	210	64	34
16	11	176	44	27

hand, the field polynomials for the ONB2 can be computed using the following recursion:

$$\begin{aligned} f_0(x) &= 1 \\ f_1(x) &= x+1 \\ f_n(x) &= x f_{n-1}(x) + f_{n-2} \quad \text{for} \quad n \geq 2 \end{aligned} \tag{4.13}$$

More information about field polynomials of optimal normal bases can be found in [31]. The change of basis matrix which allows us to perform the conversion between optimal normal basis and polynomial basis can be calculated using the algorithms given in [14, pages 37-39].

We now give an example in order to illustrate the use of a polynomial basis inversion algorithm in an optimal normal basis. Let $A \in GF((2^n)^{11})$ be expressed in the ONB2 as

$$A = A_0\beta + A_1\beta^2 + A_2\beta^{2^2} + \cdots + A_{10}\beta^{2^{10}} ,$$

where $\gcd(n, 11) = 1$. We first compute the field polynomial for $GF((2^n)^{11})$ using the recursion (4.13), and obtain it as

$$f_{11}(x) = x^{11} + x^{10} + x^8 + x^4 + x^3 + x^2 + 1 .$$

Let the polynomial representation of A be

$$A = \bar{A}_0 + \bar{A}_1\alpha + \bar{A}_2\alpha^2 + \ldots + \bar{A}_{10}\alpha^{10} ,$$

where α is a root of $f_{11}(x)$. We then use the field polynomial and the algorithm in [14] in order the obtain the change of basis matrices between the polynomial basis and the ONB2. We summarize the final change of basis formulae below. The conversion from the ONB2 representation to the polynomial is performed using:

$$\begin{aligned}
\bar{A}_0 &= A_0 \\
\bar{A}_1 &= A_0 + A_4 + A_5 + A_6 + A_8 + A_{10} \\
\bar{A}_2 &= A_1 + A_7 + A_9 + A_{10} \\
\bar{A}_3 &= A_6 + A_8 \\
\bar{A}_4 &= A_2 + A_{10} \\
\bar{A}_5 &= A_4 + A_5 + A_{10} \\
\bar{A}_6 &= A_7 + A_9 \\
\bar{A}_7 &= A_4 + A_5 \\
\bar{A}_8 &= A_3 + A_{10} \\
\bar{A}_9 &= A_5 + A_{10} \\
\bar{A}_{10} &= A_7 + A_{10} .
\end{aligned}$$

The conversion from the polynomial representation to the ONB2 representation is performed using:

$$\begin{aligned}
A_0 &= \bar{A}_1 + \bar{A}_3 + \bar{A}_7 + \bar{A}_{10} \\
A_1 &= \bar{A}_2 + \bar{A}_6 + \bar{A}_0 \\
A_2 &= \bar{A}_4 + \bar{A}_0 \\
A_3 &= \bar{A}_8 + \bar{A}_0 \\
A_4 &= \bar{A}_7 + \bar{A}_9 + \bar{A}_0 \\
A_5 &= \bar{A}_9 + \bar{A}_0
\end{aligned}$$

$$
\begin{aligned}
A_6 &= \bar{A}_5 + \bar{A}_7 + \bar{A}_0 \\
A_7 &= \bar{A}_{10} + \bar{A}_0 \\
A_8 &= \bar{A}_3 + \bar{A}_5 + \bar{A}_7 + \bar{A}_0 \\
A_9 &= \bar{A}_6 + \bar{A}_{10} + \bar{A}_0 \\
A_{10} &= \bar{A}_0 \, .
\end{aligned}
$$

4.6 Implementation Results and Conclusions

In order to test the proposed methods, we wrote test routines in the C language, and obtained the timing results using the Microsoft Visual C++ 5.0 on a 450-MHz Pentium II based PC running Windows NT 4.0. All timing values given in Tables 4.2, 4.4, 4.5, 4.6 and Figures 4.1, 4.2, 4.3, are in microseconds.

Table 4.5. The timings in microseconds for the ONB1

n	m	nm	Squaring	Multiplication	Inversion
13	18	234	0.96	13.21	46
	28	364	1.33	28.50	98
	36	468	1.60	44.80	149
15	28	420	1.65	32.26	114

The timing results for the polynomial basis implementation of the composite fields are given in Table 4.2. The squaring, multiplication, and the inversion methods are the same as in [57] except that we allow irreducible trinomials and pentanomials while the methods in [57] cover only irreducible trinomials. Table 4.2 clearly illustrates the advantage of using the subfields other than $GF(2^{16})$. For example, the multiplication operation in $GF((2^{16})^{13})$ takes 11.0 microseconds, while it takes only 9.6 microseconds for $GF((2^{13})^{16})$. Since the lookup tables for $n = 13$

Table 4.6. The timings in microseconds for the ONB2

n	m	nm	Squaring	Multiplication	Inversion
13	14	182	0.73	12.37	29
	18	234	0.86	21.93	45
	23	299	1.04	26.81	70
	29	377	1.23	43.00	106
	30	390	1.27	43.18	113
	33	429	1.38	55.21	135
	35	455	1.45	61.82	153
14	23	322	1.18	28.45	73
	29	406	1.42	46.34	118
	33	462	1.57	59.22	148
15	11	165	0.78	8.31	22
	14	210	0.89	13.70	34
	23	345	1.28	35.27	76
	26	390	1.48	36.75	94
	29	435	1.51	47.55	124
16	11	176	1.01	10.38	27
	23	368	1.73	38.16	100
	29	464	2.12	59.71	154

are smaller than those for $n = 16$, the memory access times are shorter, and thus, the multiplication is faster.

The squaring algorithm for the ONB1 and ONB2 was described in§4.5.1. Its timing values are tabulated in Tables 4.5 and 4.6. The multiplication algorithms for the ONB1 and ONB2 were described in §4.5.2 and §4.5.3, respectively, which are based on the previously developed methods reported in [27, 54]. Both algorithms use a permutation to convert the elements expressed in the optimal normal basis to the polynomial basis (actually, to a basis similar to the polynomial basis), and perform the multiplication operation in the polynomial basis using these specialized algorithms, and the convert the result back to the optimal normal basis.

Figure 4.1. Squaring timings in microseconds.

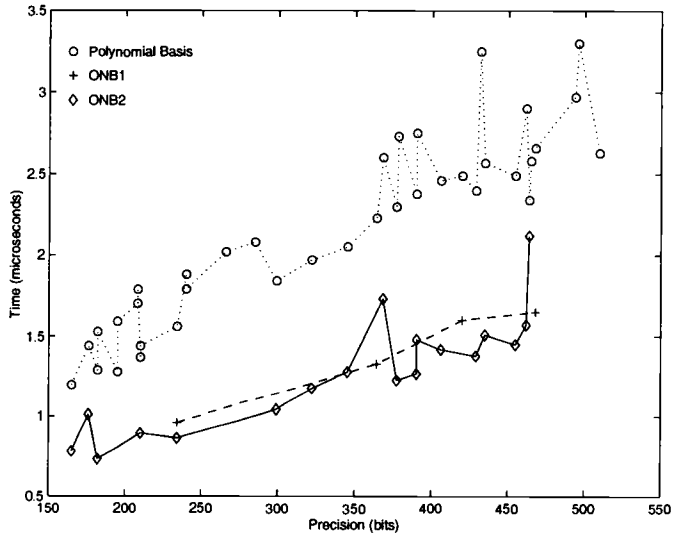


Figure 4.2. Multiplication timings in microseconds.

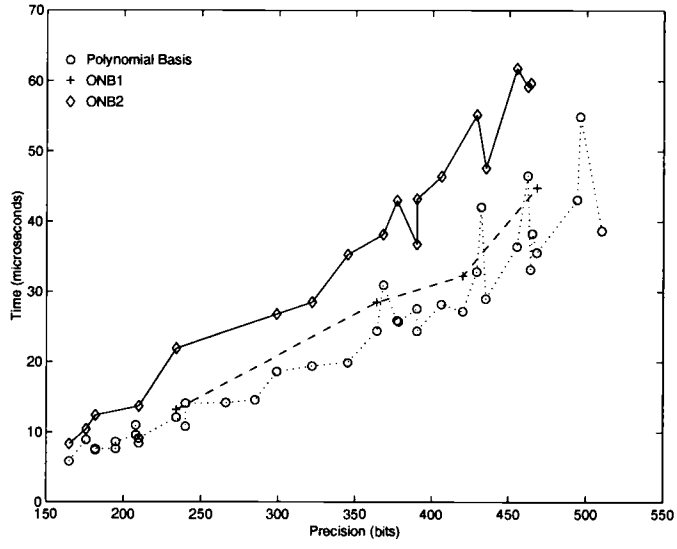
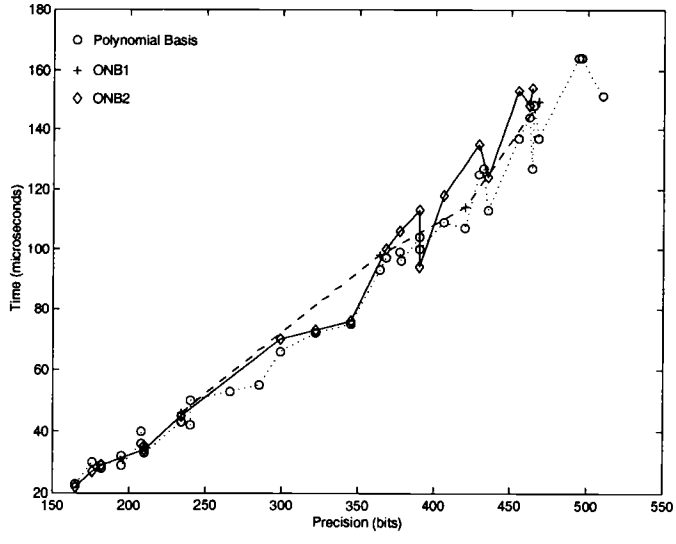


Figure 4.3. Inversion timings in microseconds.



The inversion method we proposed in §4.5.4 for the composite fields is an alternative to the well-known Itoh-Tsujii [16] algorithm. Our method performs the inversion of an element expressed in ONB1 or ONB2 by first converting it to the polynomial basis using the field polynomial of the optimal normal basis. It then uses the extended Euclidean algorithm to obtain the inverse of the given element in the polynomial basis. The result is converted back to the optimal normal basis. The field polynomial is an all-one-polynomial for the ONB1 and a random polynomial for the ONB2. The type of the field polynomial makes a difference only in the conversions between the optimal normal basis and the polynomial basis: there will be more additions (XORs) for the ONB2 case in general. The performance of the extended Euclidean algorithm is the same for any field polynomial. We compare the timings of the Itoh-Tsujii algorithm and the extended Euclidean algorithm in Table 4.4, which shows that even though the overhead of the conversions slows down the inversion operation, the extended Euclidean algorithm as proposed for the composite fields is still faster than Itoh-Tsujii algorithm; in some cases it is more than twice faster. Thus, we used the extended Euclidean in the rest of our implementation. The inversion timings given in Tables 4.5 and 4.6 are obtained using the extended Euclidean algorithm.

In Figures 4.1, 4.2, and 4.3, we illustrate the squaring, multiplication, and inversion timings together in order to compare the performance of these three bases. We clearly see from these figures that the squaring operation in the ONB1 and ONB2 is much faster than in the polynomial basis since the reduction is avoided. On the other hand, the multiplication operation is slower for the ONB1 and ONB2, however, it is not significantly slower. When we compare the inversion operation in these three bases, we notice that their timings are very close to one another. Between the ONB1 and ONB2, we also see that the ONB2 is more advantageous since it provides more composite fields in the specified range $[160, 512]$ as it can be seen in Table 4.3.

Chapter 5
Efficient Conversion Algorithms for Binary and Composite Fields

There has been a growing interest to develop hardware and software methods for implementing the finite field arithmetic operations particularly for cryptographic applications [48, 57, 43, 52, 58, 59]. In order to obtain efficient implementations, the computations are often performed in bases other than the standard polynomial basis for the field $GF(2^k)$. Thus, we are often faced with the basis conversion problems between two different implementations of the same field. For example, two such conversion problems were addressed recently [21, 20, 19].

A particularly interesting case occurs when the field $GF(2^k)$ is a composite field, i.e., k is not a prime and can be written as $k = nm$. It has been observed that efficient hardware and software implementations can be obtained for such fields [57, 43]. Thus, instead of performing the computations in the binary field, it is more efficient to implement the composite field to perform the computations. This methodology requires that we construct the composite field by suitably selecting n and m, and also by finding an irreducible polynomial to generate the field $GF((2^n)^m)$. Furthermore, efficient methods are needed for conversion of elements between the binary and composite fields. The general methodology for constructing composite fields is well established [4]. The conversion problem between the composite and binary fields and the selection of a suitable primitive element was addressed [41]. In this work, Paar derives the conversion matrix between the fields $GF(2^k)$ and $GF((2^n)^m)$ which are already known (fixed) by their generating polynomials [41].

In this chapter, we examine a slightly different problem: we construct a composite field $GF((2^n)^m)$ given the binary field $GF(2^k)$, assuming the generating

polynomial of the composite field was not fixed or given a priori. We introduce practical algorithms for constructing the field $GF((2^n)^m)$ and for obtaining the conversion matrix given the binary field $GF(2^k)$. We also give efficient conversion algorithms for the case $\gcd(n, m) = 1$, which do not require the storage of the conversion matrix. Our approach requires the use of a primitive element in $GF(2^k)$ in order to construct the composite field $GF((2^n)^m)$. However, variations are possible, for example, a non-primitive element can also be used. Furthermore, we show how to construct the composite field with a special irreducible generating polynomial, e.g., a trinomial, a pentanomial, or an equally-spaced-polynomial.

5.1 Fundamentals

Let $GF(2^k)$ denote the binary extension field defined over the prime field $GF(2)$. In order to construct $GF(2^k)$ and represent its elements, we need an irreducible polynomial $p(x)$ of degree k whose coefficients are in $GF(2)$. If α is a root of $p(x)$, then the set $B_1 = \{1, \alpha, \alpha^2, \ldots, \alpha^{k-1}\}$ forms a basis for the field $GF(2^k)$. An element A of $GF(2^k)$ can be expressed as $A = \sum_{i=0}^{k-1} a_i \alpha^i$, where $a_i \in GF(2)$ for $i = 0, 1, \ldots, k-1$. The row vector $(a_0, a_1, \ldots, a_{k-1})$ is called the representation of the element A in the basis B_1. Once the basis is selected, the rules for the field operations, e.g., addition, multiplication, and inversion, can be derived.

There are various ways to represent the elements of $GF(2^k)$, depending on the choice of the basis or the particular construction method. If k is the product of two integers as $k = mn$, then it is possible to derive a different representation method by defining $GF(2^k)$ over the ground field $GF(2^n)$. An extension field defined over a subfield of $GF(2^k)$ other than the prime field $GF(2)$ is known as the composite field. We will use $GF((2^n)^m)$ to denote the composite field. Since there is only one field with 2^k elements, both the binary and the composite fields refer to this same field. However, their representation methods are different, and it is possible to obtain one representation from the other.

The field $GF(2^n)$ over which the composite field is defined is called the ground field. Since the composite field is defined over $GF(2^n)$, we need an irreducible polynomial of degree m with coefficients in the ground field $GF(2^n)$. Let $q(x)$ be an irreducible polynomial of degree m defined over $GF(2^n)$. If β is a root of $q(x)$, then the set $B_2 = \{1, \beta, \beta^2, \ldots, \beta^{m-1}\}$ forms a basis for $GF((2^n)^m)$. An element $A \in GF((2^n)^m)$ can be written as $A = \sum_{i=0}^{m-1} a'_i \beta^i$, where $a'_i \in GF(2^n)$. The row vector $(a'_0, a'_1, \ldots, a'_{m-1})$ is the composite field representation of A in the basis B_2. The coefficients in the composite field representation are in the ground field $GF(2^n)$, and thus, we need to be able to perform field operations in $GF(2^n)$ in order to perform field operations in $GF((2^n)^m)$. Therefore, we need an irreducible polynomial $r(x)$ of degree n over $GF(2)$ in order to construct the ground field $GF(2^n)$. If γ is a root of $r(x)$, then the set $B_3 = \{1, \gamma, \gamma^2, \ldots, \gamma^{n-1}\}$ is a basis for $GF(2^n)$, thus, an element $a \in GF(2^n)$ can be written as $a = \sum_{i=0}^{n-1} \bar{a}_i \gamma^i$, where $\bar{a}_i \in GF(2)$. The row vector $(\bar{a}_0, \bar{a}_1, \ldots, \bar{a}_{n-1})$ represents the element $a \in GF(2^n)$ in the basis B_3.

In some cases, it is important to distinguish the elements of a subfield within a field or compute the order of an element. We will make use of the following results from [31, 29].

- If $\alpha \in GF((2^n)^m)$, then $\alpha^r \in GF(2^n)$, where $r = (2^{nm} - 1)/(2^n - 1)$.
- Let the order of α be denoted by $\text{ord}(\alpha)$. Then, the order of α^r is found as

$$\text{ord}(\alpha^r) = \frac{\text{ord}(\alpha)}{\gcd(r, \text{ord}(\alpha))} \quad .$$

5.2 Construction of the Composite Field

The proposed construction method depends on the availability of a primitive element in $GF(2^k)$. We make use of the following theorem.

Theorem 5.1 *Let α be a primitive element in $GF((2^n)^m)$, then $\gamma = \alpha^r$ is a primitive element in $GF(2^n)$, where $r = (2^{nm} - 1)/(2^n - 1)$.*

Proof 1 Since α is a primitive element in $GF((2^n)^m)$, its order is $\text{ord}(\alpha) = 2^{nm}-1$. The order of the element $\gamma = \alpha^r$ can be computed as

$$\text{ord}(\gamma) = \text{ord}(\alpha^r) = \frac{\text{ord}(\alpha)}{\text{gcd}(\text{ord}(\alpha), r)} .$$

Substituting $\text{ord}(\alpha) = 2^{nm} - 1$ and $r = \frac{2^{nm}-1}{2^n-1}$ in this expression, we obtain

$$\text{ord}(\gamma) = \frac{2^{nm} - 1}{\text{gcd}(2^{nm} - 1, \frac{2^{nm}-1}{2^n-1})} = \frac{2^{nm-1}}{\frac{2^{nm}-1}{2^n-1}} = 2^n - 1 .$$

Since $\text{ord}(\gamma) = 2^n - 1$, we conclude that γ is a primitive element in the subfield $GF(2^n)$. □

Let $GF((2^n)^m)$ be an extension field of $GF(2^n)$ and $\alpha \in GF((2^n)^m)$. The set of the elements

$$\mathcal{C} = \{\alpha, \alpha^{2^n}, \alpha^{2^{2n}}, \ldots, \alpha^{2^{(m-1)n}}\} \tag{5.1}$$

is called the *conjugates* of α with respect to $GF(2^n)$. The conjugates of α are not necessarily distinct elements of $GF((2^n)^m)$. Every element $\alpha \in GF((2^n)^m)$ is associated with a monic irreducible polynomial whose coefficients are in one of the subfields of $GF((2^n)^m)$. This polynomial is called the *minimal polynomial* of α and will be denoted by $m_\alpha(x)$. The conjugates of $\alpha \in GF((2^n)^m)$ are distinct if and only if the minimal polynomial of α over $GF(2^n)$ is of degree m. If all conjugates of α with respect to $GF(2^n)$ are distinct, then the minimal polynomial of α can be given as

$$m_\alpha(x) = (x + \alpha)(x + \alpha^{2^n})(x + \alpha^{2^{2n}}) \cdots (x + \alpha^{2^{(m-1)n}}) . \tag{5.2}$$

The polynomial $m_\alpha(x)$ is an irreducible polynomial of degree m with coefficients in $G(2^n)$. These definitions of the conjugates and the minimal polynomial of an element of the composite field are given with respect to a subfield of the composite field. If the prime field $GF(2)$ is taken as the the subfield, then we obtain the definitions of the conjugates and minimal polynomial of an element in the binary field $GF(2^k)$. For example, let $GF(2^k)$ be the binary field with $k = nm$ and α be

primitive element in $GF(2^k)$, then the conjugates of α and its minimal polynomial can be given as

$$C' = (\alpha, \alpha^2, \alpha^{2^2}, \ldots, \alpha^{2^{(k-1)}}) , \tag{5.3}$$

$$m'_\alpha(x) = (x+\alpha)(x+\alpha^2)(x+\alpha^{2^2})\cdots(x+\alpha^{2^{(k-1)}}) . \tag{5.4}$$

The polynomials $m_\alpha(x)$ and $m'_\alpha(x)$ are the minimal polynomials of the same element α with respect to the subfields $GF(2^n)$ and $GF(2)$, respectively. If α is a primitive element, then its conjugates with respect to both subfields are distinct. Therefore, $m_\alpha(x)$ is an irreducible polynomial of degree m whose coefficients are from $GF(2^n)$. Similarly, $m'_\alpha(x)$ is an irreducible polynomial of degree k whose coefficients are from $GF(2)$. Given the binary field $GF(2^k)$ and the minimal polynomial $m'_\alpha(x)$, we use $m_\alpha(x)$ to construct the field $GF((2^n)^m)$.

5.3 Derivation of the Conversion Matrix

In this section, we show the derivation of the general conversion matrix from the composite field to the binary field representation. Let $p(x)$ be a degree-k primitive polynomial defined over $GF(2)$ and α be a root of $p(x)$. We construct the field $GF(2^k)$ using $p(x)$, where α is used to obtain the basis

$$B_1 = \{1, \alpha, \alpha^2, \ldots, \alpha^{k-1}\} .$$

Here $p(x)$ is the minimal polynomial of α with respect to $GF(2)$. To obtain the composite field representation, we will obtain the minimal polynomial of α with respect to $GF(2^n)$. We denote this polynomial by $q(x)$, which is given as

$$q(x) = (x+\alpha)(x+\alpha^{2^n})(x+\alpha^{2^{2n}})\cdots(x+\alpha^{2^{(m-1)n}}) . \tag{5.5}$$

We use $q(x)$ to construct the field $GF((2^n)^m)$ defined over $GF(2^n)$, where the basis is

$$B_2 = \{1, \alpha, \alpha^2, \ldots, \alpha^{m-1}\} .$$

Using the bases B_1 and B_2, we obtain two different representations of the element A as

$$\text{Basis } B_1: \quad A = \sum_{i=0}^{k-1} a_i \alpha^i , \quad a_i \in GF(2) .$$

$$\text{Basis } B_2: \quad A = \sum_{j=0}^{m-1} a'_j \alpha^j , \quad a'_j \in GF(2^n) .$$

To obtain the conversion rule between these two representations of the field, we construct the basis of representation of the ground field $GF(2^n)$ in a special way. To obtain a basis, we select the constant coefficient γ of the minimal polynomial $q(x)$ with respect to the field $GF(2^n)$. Since γ is a coefficient of the minimal polynomial, it belongs to $GF(2^n)$. The explicit connection between γ and α is as follows: $\gamma = \alpha^r$, where r is given as

$$r = \frac{2^{nm} - 1}{2^n - 1} = 1 + 2^n + 2^{2n} + 2^{3n} + \ldots + 2^{(m-1)n} . \tag{5.6}$$

Thus, the basis to represent the subfield $GF(2^n)$ is given as $B_3 = \{1, \gamma, \gamma^2, \ldots, \gamma^{n-1}\}$. Therefore, a'_js are represented using the basis B_3 as

$$\text{Basis } B_3: \quad a'_j = \sum_{i=0}^{n-1} \bar{a}_{ji} \gamma^i , \quad \bar{a}_{ij} \in GF(2)$$

In order to obtain the conversion matrix from the composite field $GF((2^n)^m)$ to the binary field $GF(2^k)$, we write

$$A = \sum_{j=0}^{m-1} \sum_{i=0}^{n-1} \bar{a}_{ji} \gamma^i \alpha^j = \sum_{j=0}^{m-1} \sum_{i=0}^{n-1} \bar{a}_{ji} \alpha^{ri+j} . \tag{5.7}$$

Here the terms α^{ri+j} are reduced using the generating polynomial $p(x)$, and their representations in B_1 are obtained as

$$\alpha^{ri+j} = \sum_{h=0}^{k-1} t_{jih} \alpha^h , \tag{5.8}$$

where $t_{jih} \in GF(2)$ are the elements of the conversion matrix. By substituting (5.8) into (5.7), we derive the binary representation of A from its composite representation as

$$A = \sum_{h=0}^{k-1} \sum_{j=0}^{m-1} \sum_{i=0}^{n-1} \bar{a}_{ji} \, t_{jih} \, \alpha^h . \tag{5.9}$$

This sum determines the conversion matrix between two representations, as follows:

$$\left[\begin{array}{c} a_0 \\ \vdots \\ a_{n-1} \\ \hline a_n \\ \vdots \\ a_{2n-1} \\ \hline \vdots \\ \hline a_{mn-n} \\ \vdots \\ a_{mn-1} \end{array}\right] = \left[\begin{array}{c|c|c|c} T_{0,0} & T_{0,1} & \cdots & T_{0,m-1} \\ \hline T_{1,0} & T_{1,1} & \cdots & T_{1,m-1} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline T_{m-1,0} & T_{m-1,1} & \cdots & T_{m-1,m-1} \end{array}\right] \left[\begin{array}{c} \bar{a}_{00} \\ \vdots \\ \bar{a}_{0(n-1)} \\ \hline \bar{a}_{10} \\ \vdots \\ \bar{a}_{1(n-1)} \\ \hline \vdots \\ \hline \bar{a}_{(m-1)0} \\ \vdots \\ \bar{a}_{(m-1)(n-1)} \end{array}\right], \quad (5.10)$$

Each one of $T_{i,j}$ is an $n \times n$ matrix whose entries are from the field $GF(2)$. The entire T matrix is an $k \times k$ matrix with entries from $GF(2)$. Once the T matrix is obtained the conversion matrix from the binary field to the composite field can be obtained by computing T^{-1}. Both of these matrices need to be precomputed and saved.

Example: We show the construction of the conversion matrix T from the composite field $GF((2^3)^4)$ to the binary field $GF(2^{12})$. Let $GF(2^{12})$ be constructed using the primitive polynomial $p(x) = x^{12} + x^7 + x^4 + x^3 + 1$ and α be a root of $p(x)$, thus, α is a primitive element in $GF(2^{12})$. As we have shown, $\gamma = \alpha^r$ is a primitive element in the ground field $GF(2^3)$, where $r = (2^{12} - 1)/(2^3 - 1) = 585$. We construct the composite field $GF((2^3)^4)$ over the field $GF(2^3)$ using the irreducible polynomial $q(x)$ which is constructed according to Equation (5.4). The irreducible polynomial $q(x)$ is of degree 4 and its coefficients are from the ground field $GF(2^3)$, which is given as follows

$$\begin{aligned} q(x) &= (x+\alpha)(x+\alpha^{2^3})(x+\alpha^{2^6})(x+\alpha^{2^9}) \quad (5.11) \\ &= x^4 + \alpha^{1755}\, x^3 + \alpha^{2340}\, x^2 + \alpha^{585}\,. \end{aligned}$$

Note that α is in $GF(2^{12})$, however, $\alpha^r = \alpha^{585}$ is an element of $GF(2^3)$, and so are

$\alpha^{1755} = (\alpha^{585})^3$ and $\alpha^{2340} = (\alpha^{585})^4$. Furthermore, we have $(\alpha^{585})^7 = (\alpha^{1755})^7 = (\alpha^{2340})^7 = 1$. In order to represent the elements of the ground field $GF(2^3)$, we use the constant term in $q(x)$ as the basis element, which is $\gamma = \alpha^{585}$. An element A is expressed in basis B_2 as

$$A = a'_0 + a'_1\alpha + a'_2\alpha^2 + a'_3\alpha^3 \ , \tag{5.12}$$

where $a'_j \in GF(2^3)$. We can express a'_j in $GF(2^3)$ using $\gamma = \alpha^{585}$ as the basis element

$$a'_j = \bar{a}_{j0} + \bar{a}_{j1}\gamma + \bar{a}_{j2}\gamma^2 \ = \ \bar{a}_{j0} + \bar{a}_{j1}\alpha^{585} + \bar{a}_{j2}\alpha^{1170} \ , \tag{5.13}$$

where $\bar{a}_{ji} \in GF(2)$ for $j = 0, 1, 2, 3$ and $i = 0, 1, 2$. Therefore, the representation of A in the composite field is found as

$$\begin{aligned} A \ = \ & \bar{a}_{00} + \bar{a}_{01}\alpha^{585} + \bar{a}_{02}\alpha^{1170} + \bar{a}_{10}\alpha + \bar{a}_{11}\alpha^{586} + \bar{a}_{12}\alpha^{1171} + \\ & \bar{a}_{20}\alpha^2 + \bar{a}_{21}\alpha^{587} + \bar{a}_{22}\alpha^{1172} + \bar{a}_{30}\alpha^3 + \bar{a}_{31}\alpha^{588} + \bar{a}_{32}\alpha^{1173} \ . \end{aligned} \tag{5.14}$$

The next step is to reduce the terms α^{585i+j} for $j = 0, 1, 2, 3$ and $i = 0, 1, 2$ using the generating polynomial $p(x) = x^{12} + x^7 + x^4 + x^3 + 1$. This will give us α terms in the above expression with exponents between 0 and 11. A term of the form α^{585i+j} is reduced modulo $p(x)$ by successively using the relation $\alpha^{12} = \alpha^7 + \alpha^4 + \alpha^3 + 1$. We have obtained all higher powers of α in Equation (5.14) using a simple Maple code, as follows:

$$\begin{aligned} \alpha^{585} &= \alpha^{11} + \alpha^{10} + \alpha^8 + \alpha^7 + \alpha^5 + \alpha^2 + 1 \ , \\ \alpha^{1170} &= \alpha^{10} + \alpha^9 + \alpha^6 + \alpha^4 + \alpha^3 + \alpha + 1 \ , \\ \alpha^{586} &= \alpha^{11} + \alpha^9 + \alpha^8 + \alpha^7 + \alpha^6 + \alpha^4 + \alpha + 1 \ , \\ \alpha^{1171} &= \alpha^{11} + \alpha^{10} + \alpha^7 + \alpha^5 + \alpha^4 + \alpha^2 + \alpha \ , \\ \alpha^{587} &= \alpha^{10} + \alpha^9 + \alpha^8 + \alpha^5 + \alpha^4 + \alpha^3 + \alpha^2 + \alpha + 1 \ , \\ \alpha^{1172} &= \alpha^{11} + \alpha^8 + \alpha^7 + \alpha^6 + \alpha^5 + \alpha^4 + \alpha^2 + 1 \ , \\ \alpha^{588} &= \alpha^{11} + \alpha^{10} + \alpha^9 + \alpha^6 + \alpha^5 + \alpha^4 + \alpha^3 + \alpha^2 + \alpha \ , \\ \alpha^{1173} &= \alpha^9 + \alpha^8 + \alpha^6 + \alpha^5 + \alpha^4 + \alpha + 1 \ . \end{aligned}$$

By substituting the above terms in expression (5.14), we obtain the representation of A in the binary field $GF(2^{12})$ using basis $B_1 = (1, \alpha, \alpha^2, \ldots, \alpha^{11})$ as

$$\begin{aligned} A &= a_0 + a_1\alpha + a_2\alpha^2 + a_3\alpha^3 + a_4\alpha^4 + a_5\alpha^5 + a_6\alpha^6 + \\ &\quad a_7\alpha^7 + a_8\alpha^8 + a_9\alpha^9 + a_{10}\alpha^{10} + a_{11}\alpha^{11} \ . \end{aligned}$$

The relationship between the terms a_h for $h = 0, 1, \ldots, 11$ and $\bar{a}_{ji}$ for $j = 0, 1, 2, 3$ and $i = 0, 1, 2$ determines the elements t_{jih} of the conversion matrix T. For example, the first row of the matrix T is obtained by gathering the constant terms in the right hand side of (5.14) after the substitution, which gives the constant coefficient in the left hand side, i.e., the term a_0. A simple inspection shows that

$$a_0 = \bar{a}_{00} + \bar{a}_{01} + \bar{a}_{02} + \bar{a}_{11} + \bar{a}_{21} + \bar{a}_{22} + \bar{a}_{32} \ ,$$

which determines the first row of T. Similarly, a_1 is obtained by summing the coefficients of α as

$$a_1 = \bar{a}_{02} + \bar{a}_{10} + \bar{a}_{11} + \bar{a}_{12} + \bar{a}_{21} + \bar{a}_{31} + \bar{a}_{32} \ ,$$

which determines the next row of T. The remaining terms a_i for $i = 2, 3, \ldots, 11$ are obtained similarly, i.e., by gathering the coefficients of α^i for $i = 2, 3, \ldots, 11$, respectively. Therefore, we obtain the 12×12 dimensional matrix T as follows:

$$\left[\begin{array}{c} a_0 \\ a_1 \\ a_2 \\ \hline a_3 \\ a_4 \\ a_5 \\ \hline a_6 \\ a_7 \\ a_8 \\ \hline a_9 \\ a_{10} \\ a_{11} \end{array}\right] = \left[\begin{array}{cccccc|cccccc|cccccc|cccccc} 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ \hline 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ \hline 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ \hline 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{array}\right] \left[\begin{array}{c} \bar{a}_{00} \\ \bar{a}_{01} \\ \bar{a}_{02} \\ \hline \bar{a}_{10} \\ \bar{a}_{11} \\ \bar{a}_{12} \\ \hline \bar{a}_{20} \\ \bar{a}_{21} \\ \bar{a}_{22} \\ \hline \bar{a}_{30} \\ \bar{a}_{31} \\ \bar{a}_{32} \end{array}\right] . \qquad (5.15)$$

This matrix gives the representation of an element in the binary field $GF(2^{12})$ given its representation in the composite field $GF((2^3)^4)$. The inverse transformation, i.e., the conversion from $GF(2^{12})$ to $GF((2^3)^4)$, requires the computation of T^{-1}.

5.4 Special Case of $\gcd(n, m) = 1$

The arithmetic operations in $GF((2^n)^m)$ can be implemented much faster in software [57] or using fewer gates in hardware [43] if the degree-m irreducible polynomial is selected such that its coefficients are in $GF(2)$ instead of $GF(2^n)$. It is well-known [29] that an irreducible polynomial over $GF(2)$ of degree m remains irreducible over $GF(2^n)$ if and only if $\gcd(n, m) = 1$. Therefore, one can select an irreducible polynomial of degree m with coefficients from $GF(2)$ rather than $GF(2^n)$ if n and m are relatively prime. Also, we can use special irreducible polynomials, for example, trinomials or all-one-polynomials, to further accelerate the operations.

In this section, we show the construction of the conversion matrix from the composite field $GF((2^n)^m)$ to the binary field $GF(2^k)$ for the case of $\gcd(n, m) = 1$. In order to accomplish this task, we need to construct a primitive polynomial over $GF(2^n)$ with coefficients from $GF(2)$. Let $p(x)$ be a degree-k primitive polynomial defined over $GF(2)$, and α be a root of $p(x)$. We define $\beta = \alpha^s$ such that

$$s = \frac{2^{nm} - 1}{2^m - 1} = 1 + 2^m + 2^{2m} + 2^{3m} + \ldots + 2^{(n-1)m} \ . \tag{5.16}$$

Note that the element $\beta = \alpha^s$ is the constant term of the minimal polynomial of α with respect to $GF(2^m)$, and thus, it also belongs to $GF(2^m)$. We then construct the minimal polynomial of β with respect to $GF(2^n)$ as

$$m_\beta(x) = (x + \beta)(x + \beta^{2^n})(x + \beta^{2^{2n}}) \cdots (x + \beta^{2^{(m-1)n}}) \ . \tag{5.17}$$

We have the following theorem regarding the reduction of $m_\beta(x)$ given above.

Theorem 5.2 *The minimal polynomial $m_\beta(x)$ given by (5.17) is equivalent to*

$$m_\beta(x) = (x + \beta)(x + \beta^2)(x + \beta^{2^2}) \cdots (x + \beta^{2^{(m-1)}}) \ . \tag{5.18}$$

Proof 2 Since $\beta \in GF(2^m)$, we have $\beta^{2^m-1} = 1$. Therefore, the terms of the form $\beta^{2^{in}}$ for $i = 0, 1, \ldots, (m-1)$ can be reduced using the identity $\beta^{2^m-1} = 1$. This reduction is equivalent the reduction of the set $\{1, 2^n, 2^{2n}, \ldots, 2^{(m-1)n}\}$ modulo $2^m - 1$. It turns out that the set $\{1, 2^n, 2^{2n}, \ldots, 2^{(m-1)n}\}$ is equivalent to the set $\{1, 2, 2^2, \ldots, 2^{(m-1)}\}$ modulo (2^m-1). This is easily proven by noticing the following identities:

- If $in < m$, we have $2^{in} = 2^{in} \bmod (2^m - 1)$.
- If $in = m$, we have $2^m = 1 = 2^0 \bmod (2^m - 1)$
- If $in > m$, we have $2^{in} = 2^{jm+u}$ for some j and $0 \leq u < m$. Therefore, we have the identity $2^{jm+u} = (2^m)^j 2^u = 2^u \bmod (2^m - 1)$.

It follows from these identities that $u = in \bmod m$ in all three cases. In other words, we have

$$2^{in} = 2^{in \bmod m} \bmod (2^m - 1) \ . \tag{5.19}$$

In order to prove that unique powers of 2 are generated as $\{2^0, 2^1, 2^2, \ldots, 2^{m-1}\}$, we notice that the product $in \bmod m$ for $i = 0, 1, \ldots, m-1$ generates the unique residue class mod m if and only if $\gcd(n, m) = 1$. Therefore, the minimal polynomial of β with respect to $GF(2^n)$ can be written as

$$m_\beta(x) = (x + \beta)(x + \beta^2)(x + \beta^{2^2}) \cdots (x + \beta^{2^{(m-1)}}) \ .$$

□

The polynomial $q(x) = m_\beta(x)$ given by (5.17) is exactly of the same form as the minimal polynomial of β with respect to the field $GF(2)$, and therefore, its coefficents belong to $GF(2)$. We note that $q(x)$ is irreducible over $GF(2)$ and also $GF(2^n)$ whenever $\gcd(n, m) = 1$. We use $q(x)$ to construct the composite representation for $GF((2^n)^m)$. An element of $GF((2^n)^m)$ can be written as

$$A = \sum_{j=0}^{m-1} a'_j \beta^j \ , \tag{5.20}$$

where $a'_j \in GF(2^n)$. To represent the subfield $GF(2^n)$, similar to the previous construction, we choose the basis generated by $\gamma = \alpha^r$, where $r = \frac{2^{nm}-1}{2^n-1}$, and obtain the representation of a'_j as

$$a'_j = \sum_{i=0}^{n-1} \bar{a}_{ji}\gamma^i \ , \tag{5.21}$$

where $\bar{a}_{ji} \in GF(2)$. By combining these two representations, we obtain

$$A = \sum_{j=0}^{m-1}\sum_{i=0}^{n-1} \bar{a}_{ji}\gamma^i\beta^j = \sum_{j=0}^{m-1}\sum_{i=0}^{n-1} \bar{a}_{ji}\alpha^{(ri+sj)} \ , \tag{5.22}$$

where $\bar{a}_{ji} \in GF(2)$. We reduce the terms $\alpha^{(ri+sj)}$ using the generating polynomial $p(x)$, and obtain their representation the basis $\{1, \alpha, \alpha^2, \ldots, \alpha^{k-1}\}$ as

$$\alpha^{ri+sj} = \sum_{h=0}^{k-1} t_{jih}\alpha^h \ , \tag{5.23}$$

where $t_{jih} \in GF(2)$ are the elements of the conversion matrix. By substitution, we derive the binary representation of A from its composite representation

$$A = \sum_{h=0}^{k-1}\sum_{j=0}^{m-1}\sum_{i=0}^{n-1} \bar{a}_{ji}\ t_{jih}\ \alpha^h \ . \tag{5.24}$$

This sum gives the conversion matrix T between two representations, similar to Equation (5.9).

Example: Let $GF(2^{12})$ be constructed using the primitive polynomial $p(x) = x^{12}+x^7+x^4+x^3+1$ and α be a root of $p(x)$, thus, α is a primitive element in $GF(2^{12})$. We will calculate the conversion matrix from the composite field $GF((2^3)^4)$ to the binary field $GF(2^{12})$. We define $\beta = \alpha^s$, where

$$s = \frac{2^{nm}-1}{2^m-1} = \frac{2^{12}-1}{2^4-1} = \frac{4095}{15} = 273 \ .$$

The element $\beta = \alpha^{273}$ is the constant term of the minimal polynomial of α with respect to $GF(2^4)$, and thus also belongs to $GF(2^4)$. The minimal polynomial of β with respect to $GF(2^3)$ is written as

$$m_\beta(x) = (x+\beta)(x+\beta^{2^3})(x+\beta^{2^6})(x+\beta^{2^9}) \ . \tag{5.25}$$

This polynomial will be reduced using the identity $\beta^{2^m-1} = \beta^{15} = 1$. When we reduce the set $\{1, 2^3, 2^6, 2^9\}$ modulo $(2^4 - 1)$, we obtain $\{1, 8, 4, 2\} = \{1, 2, 2^2, 2^3\}$. Therefore, $m_\beta(x)$ can be written as

$$m_\beta(x) = (x + \beta)(x + \beta^2)(x + \beta^{2^2})(x + \beta^{2^3}) \ , \tag{5.26}$$

After substituting with $\beta = \alpha^{273}$ and multiplying out, we obtain $q(x) = m_\beta(x)$ as follows:

$$\begin{aligned} q(x) \ = \ & x^4 + (\alpha^{273} + \alpha^{546} + \alpha^{1092} + \alpha^{2184})\ x^3 \\ & +(\alpha^{819} + \alpha^{1365} + \alpha^{1638} + \alpha^{2457} + \alpha^{2730} + \alpha^{3276})\ x^2 \\ & +(\alpha^{1911} + \alpha^{3003} + \alpha^{3549} + \alpha^{3822})\ x + \alpha^{4095} \ . \end{aligned} \tag{5.27}$$

We reduce the above polynomial using the identity $\alpha^{12} = \alpha^7 + \alpha^4 + \alpha^3 + 1$, which gives us the simple irreducible polynomial

$$q(x) = x^4 + x^3 + 1 \ , \tag{5.28}$$

whose coefficients are from $GF(2)$ rather than $GF(2^3)$, as expected. The irreducible polynomial $q(x)$ is used to construct the composite field $GF((2^3)^4)$. An element of $GF((2^3)^4)$ is written as

$$A = a'_0 + a'_1\beta + a'_2\beta^2 + a'_3\beta^3 = a'_0 + a'_1\alpha^{273} + a'_2\alpha^{546} + a'_3\alpha^{819} \ ,$$

where $a'_j \in GF(2^3)$. We represent the subfield $GF(2^3)$ using the basis generated by $\gamma = \alpha^r$, where $r = (2^{12} - 1)/(2^3 - 1) = 585$ as follows

$$a'_j = \bar{a}_{j0} + \bar{a}_{j1}\gamma + \bar{a}_{j2}\gamma^2 = \bar{a}_{j0} + \bar{a}_{j1}\alpha^{585} + \bar{a}_{j2}\alpha^{1170} \ ,$$

where $\bar{a}_{ji} \in GF(2)$ for $j = 0, 1, 2, 3$ and $i = 0, 1, 2$. By combining these two representations, we obtain

$$\begin{aligned} A \ = \ & \bar{a}_{00} + \bar{a}_{01}\alpha^{585} + \bar{a}_{02}\alpha^{1170} + \bar{a}_{10}\alpha^{273} + \bar{a}_{11}\alpha^{858} + \bar{a}_{12}\alpha^{1443} + \\ & \bar{a}_{20}\alpha^{546} + \bar{a}_{21}\alpha^{1131} + \bar{a}_{22}\alpha^{1716} + \bar{a}_{30}\alpha^{819} + \bar{a}_{31}\alpha^{1404} + \bar{a}_{32}\alpha^{1989} \ . \end{aligned} \tag{5.29}$$

The next step is to reduce the terms $\alpha^{585i+273j}$ for $j = 0, 1, 2, 3$ and $i = 0, 1, 2$ using the generating polynomial $p(x) = x^{12} + x^7 + x^4 + x^3 + 1$. This will give us α terms in the above expression with exponents between 0 and 11. A term of the form $\alpha^{585i+273j}$ is reduced modulo $p(x)$ by successively using the relation $\alpha^{12} = \alpha^7 + \alpha^4 + \alpha^3 + 1$. We have obtained all higher powers of α in (5.29) using a simple Maple code, as follows:

$$\begin{aligned}
\alpha^{585} &= \alpha^{11} + \alpha^{10} + \alpha^8 + \alpha^7 + \alpha^5 + \alpha^2 + 1 \ , \\
\alpha^{1170} &= \alpha^{10} + \alpha^9 + \alpha^6 + \alpha^4 + \alpha^3 + \alpha + 1 \ , \\
\alpha^{273} &= \alpha^9 + \alpha^7 + \alpha^4 + \alpha^3 + 1 \ , \\
\alpha^{858} &= \alpha^{11} + \alpha^{10} + \alpha^9 + \alpha^5 + \alpha^4 + \alpha^3 \ , \\
\alpha^{1443} &= \alpha^{11} + \alpha^{10} + \alpha^9 + \alpha^8 + \alpha^7 + \alpha^5 \ , \\
\alpha^{546} &= \alpha^{10} + \alpha^6 + \alpha^4 + \alpha^2 + \alpha + 1 \ , \\
\alpha^{1131} &= \alpha^{11} + \alpha^{10} + \alpha^9 + \alpha^7 + \alpha^6 + \alpha^5 + \alpha + 1 \ , \\
\alpha^{1716} &= \alpha^{11} + \alpha^9 + \alpha^8 + \alpha^7 + \alpha^3 + \alpha^2 \ , \\
\alpha^{819} &= \alpha^{11} + \alpha^{10} + \alpha^7 + \alpha^6 + \alpha^4 + \alpha^3 + \alpha^2 \ , \\
\alpha^{1404} &= \alpha^8 + \alpha^6 + \alpha^5 + \alpha^4 + \alpha^2 + 1 \ , \\
\alpha^{1989} &= \alpha^{11} + \alpha^{10} + \alpha^9 + \alpha^6 + \alpha^5 + \alpha^3 + \alpha^2 + \alpha + 1 \ .
\end{aligned}$$

By substituting the above terms in expression (5.29), we obtain the representation of A in the binary field $GF(2^{12})$ using basis $(1, \alpha, \alpha^2, \ldots, \alpha^{11})$ as

$$\begin{aligned}
A &= a_0 + a_1\alpha + a_2\alpha^2 + a_3\alpha^3 + a_4\alpha^4 + a_5\alpha^5 + a_6\alpha^6 + \\
&\quad a_7\alpha^7 + a_8\alpha^8 + a_9\alpha^9 + a_{10}\alpha^{10} + a_{11}\alpha^{11} \ .
\end{aligned}$$

The relationship between the terms a_h for $h = 0, 1, \ldots, 11$ and $\bar{a}_{ji}$ for $j = 0, 1, 2, 3$ and $i = 0, 1, 2$ determines the elements t_{jih} of the conversion matrix T. For example, the first row of the matrix T is obtained by gathering the constant terms in the right hand side of (5.29) after the substitution, which gives the constant coefficient in the left hand side, i.e., the term a_0. A simple inspection shows that

$$a_0 = \bar{a}_{00} + \bar{a}_{01} + \bar{a}_{02} + \bar{a}_{10} + \bar{a}_{20} + \bar{a}_{21} + \bar{a}_{31} + \bar{a}_{32} \ ,$$

which determines the first row of T. Similarly, a_1 is obtained by summing the coefficients of α as

$$a_1 = \bar{a}_{02} + \bar{a}_{20} + \bar{a}_{21} + \bar{a}_{32} \ ,$$

which determines the next row of T. The remaining terms a_i for $i = 2, 3, \ldots, 11$ are obtained similarly, i.e., by gathering the coefficients of α^i for $i = 2, 3, \ldots, 11$, respectively. We obtain the 12×12 dimensional matrix T as follows:

$$\left[\begin{array}{c} a_0 \\ a_1 \\ a_2 \\ \hline a_3 \\ a_4 \\ a_5 \\ \hline a_6 \\ a_7 \\ a_8 \\ \hline a_9 \\ a_{10} \\ a_{11} \end{array}\right] = \left[\begin{array}{ccc|ccc|ccc|ccc} 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\ \hline 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ \hline 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ \hline 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \end{array}\right] \left[\begin{array}{c} \bar{a}_{00} \\ \bar{a}_{01} \\ \bar{a}_{02} \\ \hline \bar{a}_{10} \\ \bar{a}_{11} \\ \bar{a}_{12} \\ \hline \bar{a}_{20} \\ \bar{a}_{21} \\ \bar{a}_{22} \\ \hline \bar{a}_{30} \\ \bar{a}_{31} \\ \bar{a}_{32} \end{array}\right] . \qquad (5.30)$$

This matrix gives the representation of an element in the binary field $GF(2^{12})$ given its representation in the composite field $GF((2^3)^4)$. Similarly, the conversion from the field $GF(2^{12})$ to the field $GF((2^3)^4)$ requires the computation of the inverse of this matrix.

5.5 Use of Non-Primitive Elements

The proposed method of construction of the composite field $GF((2^n)^m)$ depends on the availability of a primitive element α in $GF(2^k)$, which is the root of a degree-k primitive polynomial $p(x)$ defined over $GF(2)$. We then derive the transformation (change of basis) matrix T from $GF(2^k)$ to $GF((2^n)^m)$ using the minimal

polynomial of α with respect to $GF(2^n)$ as $q(x) = m_\alpha(x)$. A question arises about the derivation of the transformation matrix in case when a non-primitive polynomial $h(x)$ is used construct the field $GF(2^k)$. In this case, we cannot construct the composite field $GF((2^n)^m)$ properly, and obtain the transformation matrix T. Fortunately, we need not a specific primitive element, any primitive element would work. The primitive elements in a finite field are abundant, and it is easy to find one given a representation of the field $GF(2^k)$. Let $h(x)$ be a non-primitive irreducible polynomial used to construct the the binary field $GF(2^k)$, and also let σ be a root of $h(x)$. The set

$$B_0 = \{1, \sigma, \sigma^2, \ldots, \sigma^{k-1}\} \tag{5.31}$$

forms a basis for the field $GF(2^k)$. Let α be a primitive element in the field $GF(2^k)$. We can use the primitive element α to construct the composite field $GF((2^n)^m)$ properly, as in § 5.3 (or, as in § 5.4 if $\gcd(n, m) = 1$). According to § 5.3, we have the bases B_1, B_2, and B_3 as

$$\begin{aligned} B_1 &= \{1, \alpha, \alpha^2, \ldots, \alpha^{k-1}\} , \\ B_2 &= \{1, \alpha, \alpha^2, \ldots, \alpha^{m-1}\} , \\ B_3 &= \{1, \gamma, \gamma^2, \ldots, \gamma^{n-1}\} , \end{aligned}$$

where α is a primitive element in $GF(2^k)$ and $\gamma = \alpha^r$ with $r = (2^{nm} - 1)/(2^n - 1)$. We represent an element of the binary field $GF(2^k)$ using the basis B_1. On the other hand, we represent an element of $GF((2^n)^m)$ using the basis B_2, where the coefficients in this representation are represented using the basis B_3. However, since an element of $GF(2^k)$ is initially given in B_0, we need to embed the change of basis matrix from B_0 to B_1 to the final transformation matrix. According to Equation (5.7) in § 5.3, we have

$$A = \sum_{j=0}^{m-1} \sum_{i=0}^{n-1} \bar{a}_{ji} \gamma^i \alpha^j \;=\; \sum_{j=0}^{m-1} \sum_{i=0}^{n-1} \bar{a}_{ji} \alpha^{ri+j} \; .$$

Assuming the representation of the primitive element α in the basis B_0 is given, we obtain the representations of the terms α^{ri+j} in B_0 for $i = 0, 1, \ldots, n-1$ and $j = 0, 1, \ldots, m-1$, as

$$\alpha^{ri+j} = \sum_{h=0}^{k-1} \bar{t}_{ijh}\sigma^h \ . \tag{5.32}$$

This gives the modified transformation matrix based on the equation

$$A = \sum_{h=0}^{k-1} \sum_{j=0}^{m-1} \sum_{i=0}^{n-1} \bar{a}_{ji}\ \bar{t}_{jih}\ \sigma^h \ , \tag{5.33}$$

which is analogous to Equation (5.9).

5.6 Composite Fields with Special Irreducible Polynomials

In § 5.4, we constructed the composite field $GF((2^n)^m)$ for $\gcd(n, m) = 1$ in such a way that the degree-m irreducible polynomial $q(x)$ has its coefficients from $GF(2)$ rather than $GF(2^n)$. This selection yields efficient composite field arithmetic, as was demonstrated in [57]. This particular polynomial can be further specialized in the sense that it could be an irreducible trinomial, or pentanomial, or equally-spaced-polynomial (ESP), or all-one-polynomial (AOP). For instance, in the example in § 5.4, we obtained (by chance) the irreducible trinomial $x^4 + x^3 + 1$ to construct the field $GF((2^3)^4)$. Here we describe two methods by which we can select the degree-m irreducible polynomial generating the field $GF((2^n)^m)$. Let $q^*(x)$ be the irreducible degree-m polynomial of the desired form, e.g., trinomial, pentanomial, ESP, AOP, etc.

- The first method is to find a primitive element in α in $GF(2^k)$ such that

$$\begin{aligned} s &= \frac{2^{nm}-1}{2^m-1} = 1 + 2^m + 2^{2m} + 2^{3m} + \ldots + 2^{(n-1)m} \ , \\ \beta &= \alpha^s \ , \\ q(x) &= (x+\beta)(x+\beta^2)(x+\beta^{2^2})\cdots(x+\beta^{2^{(m-1)}}) \ , \\ q(x) &\stackrel{?}{=} q^*(x) \ . \end{aligned}$$

  However, this method requires that we exhaustively try all primitive elements $\alpha \in GF(2^k)$, which becomes prohibitive as k grows since it requires exponential time.

- The second method is simpler and more efficient: We go ahead with the original construction method by selecting an arbitrary primitive element α from $GF(2^k)$ and in the end obtain $q(x)$ which is an arbitrary irreducible polynomial of degree m over the field $GF(2)$ to construct the field $GF((2^n)^m)$. We then take the desired irreducible polynomial $q^*(x)$ and contsruct the change of basis matrix from the field $GF((2^n)^m)$ generated by $q(x)$ to the field $GF((2^n)^m)$ generated by $q^*(x)$. The arithmetic is performed in the latter field more efficiently due to the special structure of $q^*(x)$, and the mapped back to the former field if and when necessary.

5.7 Storage-Efficient Conversion

The proposed conversion methods between the binary and composite fields involve matrix multiplication. It also requires storing two matrices each of which has $(nm)^2$ entries. In low-cost hardware implementations, we may not have sufficient amount of memory for these matrices. Fortunately, there are other approaches which do not require the conversion matrices be stored. For exampe, Kaliski and Yin proposed storage-efficient conversion methods for the binary fields with different bases [21, 20]. Here, we take a similar approach, and introduce storage-efficient conversion algorithms between the binary and composite fields. Here we address only the case $\gcd(n, m) = 1$ since this is the most practical case for the existing applications.

According to the setup, we have two communicating parties: The first party uses the binary field and can compute only in this field, while the second one uses the composite field and can compute only in the composite field. Thus, the first party should be able to convert an element given in the second party's basis to the first party's basis using only the arithmetic which is available to the first party. Similar conditions hold for the second party.

We represent an element $\bar{A}$ of the composite field using

$$\bar{A} = (\bar{a}_{00}, \bar{a}_{01}, \ldots, \bar{a}_{0,n-1}, \bar{a}_{10}, \bar{a}_{11}, \ldots, \bar{a}_{1,n-1}, \ldots, \bar{a}_{m-1,0}, \bar{a}_{m-1,1}, \ldots, \bar{a}_{m-1,n-1}) ,$$

where $\bar{a}_{ij} \in GF(2)$ for $0 \leq i \leq n-1$ and $0 \leq j \leq m-1$. This representation can also be interpreted as

$$\bar{A} = (a'_0, a'_1, \ldots, a'_{m-1}) ,$$

where $a'_i = (a_{i,0}, a_{i,1}, \ldots, a_{i,n-1}) \in GF(2^n)$ for $0 \leq i \leq m-1$. On the other hand, an element A of the binary field is represented using the binary string $A = (a_0, a_1, \ldots, a_{mn-1})$ where $a_i \in GF(2)$ for $0 \leq i \leq mn-1$.

In order to obtain the binary representation A of $\bar{A}$, we need to know the integers r and s. Additionally, the primitive element α needs to be known. We precompute $X = \alpha^r$ and $Y = \alpha^s$, and save these values. This computation is performed using the binary field arithmetic.

	Composite To Binary
Inputs:	$\bar{A} = (\bar{a}_{00}, \bar{a}_{01}, \ldots, \bar{a}_{m-1,n-1})$
	r, s, α, $X = \alpha^r$, and $Y = \alpha^r$
Output:	$A = (a_0, a_1, \ldots, a_{mn-1})$
Step 1:	$A := 0$
Step 2:	for $j = 0$ to $m-1$
Step 3:	for $i = 0$ to $n-1$
Step 4:	if $(\bar{a}_{ji} = 1)$ then $A = A + X^i Y^j$
Step 5:	return A

The second party using the composite basis $(1, \beta, \beta^2, \ldots, \beta^{m-1})$ needs to store the primitive element α in the composite basis. We assume the primitive element α is expressed as

$$Z = \alpha = \alpha'_0 + \alpha'_1 \beta + \ldots + \alpha'_{m-1} \beta^{m-1} .$$

The primitive element in this representations needs to be precomputed (using the conversion matrix) and stored.

Binary To Composite

Inputs: $A = (a_0, a_1, \ldots, a_{mn-1})$

$Z = (\alpha'_0, \alpha'_1, \ldots, \alpha'_{m-1})$

Output: $\bar{A} = (a'_0, a'_1, \ldots, a'_{m-1})$

Step 1: $\bar{A} := 0$

Step 2: if$(a_0 = 1)$ then $a'_0 := 1$

Step 3: for $i = 1$ to $mn - 1$

Step 4: if $(a_i = 1)$ then $\bar{A} = \bar{A} + Z^i$

Step 5: return $\bar{A}$

5.8 Conclusions

We addressed a particular conversion problem in the finite fields. We construct a composite field $GF((2^n)^m)$ given the binary field $GF(2^k)$ and the integers n and m such that $k = nm$, and obtain the conversion matrices between these two representations of the same field. A variation of this idea is explored in [41], in which, given both of these fields and their field polynomials, the method searches for a suitable primitive element to obtain the conversion matrix. We are motivated from the fact that while the setup of [41] is more general, it requires exponential time since a suitable primitive element needs to be obtained. However, for many practical implementations any composite field can do the job of minimizing the time or hardware complexity.

Chapter 6
Generating Elliptic Curves of Known Order

6.1 Introduction

An important category of cryptographic algorithms is that of the elliptic curve cryptosystems defined over a finite field F_p (also denoted $GF(p)$). While there are many methods proposed for performing fast elliptic curve arithmetic, there is a paucity of efficient means for generating suitable elliptic curves. The methods proposed to date for curve generation mainly necessitate implementing complex and floating point arithmetic with high precision. However, this prevents these algorithms from being implemented on simple processors with limited amounts of memory. In [33], Miyaji proposed a practical approach to construct elliptic curves of trace 1; a class of elliptic curves which has since proved to be insecure [50]. The same idea, on the other hand, can be further utilized by allowing more general traces. (The trace meant here is the value t when the number of points of the curve is expressed as $p + 1 - t$. The term trace is related to the fact that this value is indeed the trace of a certain "Frobenius" map.) In this chapter, we present a new variant of this method to construct elliptic curves of known orders. Our variant has less computational complexity in its online implementation than that proposed in the IEEE standards [15]. Calculations show that our method is practical especially when the selection of the characteristic, p, is unimportant, as in implementations using Montgomery multiplication [34].

This chapter is organized as follows. Section 6.2 summarizes the complex multiplication curve generation method. In Section 6.3, we explain our new variant which requires less data size and computation than the general approach, while avoiding the weakness of Miyaji's method. In Section 6.4, we give some elementary heuristics

for the speed of our method to find elliptic curves with prime orders. Section 6.5 summarizes the method to construct the class polynomials, the most computationally intensive part of the CM method. In our approach, we pre-calculate a set of these and store the coefficients. Finally, in Section 6.6, we give some experimental results which indicate the efficiency of our approach.

6.2 Complex Multiplication Curve Generation Algorithm

An elliptic curve $\mathcal{E}$ defined over a finite field F_p, where $p > 3$, can be given as

$$\mathcal{E}(F_p) : y^2 = x^3 + ax + b \qquad a, b \in F_p \tag{6.1}$$

Associated with $\mathcal{E}$, there are two important quantities:
the discriminant

$$\Delta = -16(4a^3 + 27b^2) \tag{6.2}$$

and the j-invariant

$$j = 1728(4a)^3/\Delta \tag{6.3}$$

where $\Delta \neq 0$.

Lemma 6.1 *Given $j_0 \in F_p$ there is an elliptic curve, $\mathcal{E}$, defined over F_p such that $j(\mathcal{E}) = j_0$.*

An elliptic curve with a given j-invariant j_0 is constructed easily. We consider $j_0 \notin \{0, 1728\}$; these special cases are also easily handled. Let $k = j_0/(1728 - j_0)$, $j_0 \in F_p$ then the equation

$$\mathcal{E}\colon y^2 = x^3 + 3kx + 2k \tag{6.4}$$

gives an elliptic curve with j-invariant $j(\mathcal{E}) = j_0$.

Theorem 6.1 *Isomorphic elliptic curves have the same j-invariant.*

Theorem 6.2 *(Hasse) Let $\#\mathcal{E}(F_p)$ denote the number of points on the elliptic curve $\mathcal{E}(F_p)$. If $\#\mathcal{E}(F_p) = p + 1 - t$, then $|t| \leq 2\sqrt{p}$.*

Definition (Twist) : Given $\mathcal{E}$: $y^2 = x^3 + ax + b$ with $a, b \in F_p$ the *twist* of $\mathcal{E}$ by c is the elliptic curve given by

$$\mathcal{E}_c : y^2 = x^3 + ac^2x + bc^3 \tag{6.5}$$

where $c \in F_p$.

Theorem 6.3 *Let $\mathcal{E}$ be defined over F_p and its order be $\#\mathcal{E}(F_p) = p + 1 - t$. Then the order of its twist is given as*

$$\#\mathcal{E}_c(F_p) = \begin{cases} p + 1 - t & \text{if } c \text{ is square in } Z_p \\ p + 1 + t & \text{if } c \text{ is non-square in } Z_p \end{cases} \tag{6.6}$$

For the above basics of elliptic curves, we refer to [53]. The following result is based upon work of M. Deuring in the 1940s.

Theorem 6.4 *Let p be an odd prime such that*

$$4p = t^2 + Ds^2 \tag{6.7}$$

for some $t, s \in Z$. Then there is an elliptic curve $\mathcal{E}$ defined over F_p such that $\#\mathcal{E}(F_p) = p + 1 - t$.

An integer D which satisfies (6.7) for a given p is called a *CM discriminant* of p. Given such a D for a prime p, the j-invariant of the elliptic curve whose trace is t can be calculated using classfield theory. Once the j-invariant is known, the elliptic curve with $p + 1 - t$ points is easily constructed using Lemma 6.1. Actually, the method gives an elliptic curve with either $p + 1 - t$ or $p + 1 + t$ points. If the constructed elliptic curve has $p + 1 + t$ points, then one must take the twist of this elliptic curve to obtain an elliptic curve with $p + 1 - t$ points. Fortunately, it is

trivial to construct the desired curve when its twist is known, due to Theorem 6.3. This technique for constructing elliptic curves of known order is called the *Complex Multiplication* (CM) method.

A detailed explanation of CM method is given in the P1363 Standards. One can also profitably refer to [5]. We can summarize the method in the following:

1. Given a prime number p, find the smallest D in (6.7) along with t (s is not needed in the computations).

2. The orders of the curves which can be constructed are $\#\mathcal{E}(F_p) = p + 1 \pm t$. Check if one of these orders has an admissible factorization (by admissible factorization we mean a prime or nearly prime number as defined in the standards). If not, find another D and corresponding t. Repeat until an order with admissible factorization is found.

3. Construct the class polynomial $H_D(x)$ using the formulas given in the standards. (The class polynomial for a D is a fixed monic polynomial with integer coefficients. In particular, it is independent of p).

4. Find a root j_0 of $H_D(x) \pmod{p}$. This j_0 is the j-invariant of the curve to be constructed.

5. Set $k = j_0/(1728 - j_0) \pmod{p}$ and the curve is $\mathcal{E}$: $y^2 = x^3 + 3kx + 2k$.

6. Check the order of the curve. If it is not $p + 1 - t$, then construct the twist using a randomly selected non-square $c \in F_p$.

In the CM method, it is preferable to fix a prime number p, then construct the curve over F_p. It is thus advantageous to use prime numbers of special form so as to improve the efficiency of the modular arithmetic. On the other hand, the method can be efficient only when the degree of the class polynomial is small; in general, factoring a high degree polynomial is time consuming. Furthermore, the construction of the class polynomials requires multi-precision floating-point and complex number arithmetic.

6.3 A New Approach to Generating Elliptic Curves

Using special primes to increase the efficiency of the modular arithmetic might complicate the curve generation by introducing class polynomials of high degree. If one uses a method for modular arithmetic whose efficiency does not depend on the selection of p, then the CM method simplifies. Indeed, we can first determine a set $\mathcal{D}$ of discriminants D such that the corresponding class polynomials are of small degree. The idea is straightforward: Construct and store the corresponding class polynomials for D in $\mathcal{D}$ and search for primes whose CM discriminants are in this set. We thus avoid repeatedly calculating class polynomials; hence multi-precision floating and complex number arithmetic as well as the factorization of high degree class polynomials is avoided. Indeed, the original CM method as specified in the standards becomes inefficient if not impractical as the class polynomial degree becomes large.

Our new algorithm is thus:

1. Off-line: Determine a set $\mathcal{D}$ of CM discriminants such that the corresponding class numbers are small.

2. Off-line: Calculate and store the class polynomials of CM discriminants in $\mathcal{D}$.

3. Select randomly a CM discriminant D in $\mathcal{D}$ and obtain the corresponding class polynomial $H_D(x)$.

4. Search for a prime number p which satisfies the equation $4p = t^2 + Ds^2$. (First, we select random t and s values of proper sizes and then check if p is prime.)

5. Compute $u_1 = p + 1 - t$ and $u_2 = p + 1 + t$ as the orders of the elliptic curves with D and check if either of them has an admissible factorization (i.e. is a prime or nearly-prime number). If not, go to Step 4 and pick another random pair of t and s.

6. If u_1 has proper factorization set $u = u_1$, otherwise $u = u_2$.

7. Find a root j_0 of $H_D(x) \bmod p$ (this is the j-invariant of the curve).

8. Set $k = j_0/(1728 - j_0) \bmod p$ and the curve of order u_1 or u_2 is

$$\mathcal{E}_c : y^2 = x^3 + ax + b \tag{6.8}$$

where $a = 3kc^2$, $b = kc^3$ and $c \in F_p$ is randomly chosen.

9. Check the order of the curve. If it is u then stop. Otherwise, select a non-square number $e \in F_p$ and calculate the twist by e, $\mathcal{E}_e(F_p) = x^3 + ae^2 + be^3$.

At first glance, it might seem impractical to find (p, u) pairs which give elliptic curves satisfying the primality and/or near-primality conditions. However, our experiments confirm that such pairs are plentiful and can be found very quickly. (The theory of the plentitude of elliptic curves of prime and near-prime orders was pioneered by H. Lenstra, Jr. [17].)

6.4 Heuristics on Plentitude of Primes Suitable for Curve Generation

The prime Number Theorem states that for sufficiently large M, the probability of a randomly chosen integer in $[0, M]$ being prime is approximately $1/\ln M$. For our purposes, we are only interested in primes which are of the form $4p = t^2 + s^2D$. Since $p \leq M$, each pair $(s, t) \in \mathcal{Z}^2$ gives an integral lattice point inside the ellipse of equation $t^2 + s^2D = M$.

Gauss, see for example [6, p. 161], found an asymptotic formula for the number of lattice points interior to an ellipse. In our setting, this gives that the number of the lattice points (s, t) is $L = \pi M/\sqrt{D} + O(\sqrt{D})$. Of course, $(-s, -t)$ gives the same value for p as (s, t). Furthermore, our p are odd, we work with odd D and we desire the elliptic curve order $u = p + 1 \pm t$ to be prime, hence certainly odd. Thus, we will use $t = 2v + 1$, $s = 2w + 1$. We then find $L/4$ distinct values of $t^2 + s^2D$ for (s, t) interior to the ellipse.

But, $4p = t^2 + s^2 D$ expresses that p is a norm from $K = \mathcal{Q}(\sqrt{D})$. The density of prime numbers which are of this type is approximately $1/(2h_D)$, where h_D is the class number of K. (One has that the prime ideal (p) splits into the product of two principal ideals, $\Pi \cdot \Pi'$, in the ring of integers of K.) See [5, 7, 17].

From the above, we find that by choosing (v, w) at random such that $(2v + 1)^2 + D(2w + 1)^2 \leq M$, we find prime p of the correct form with a probability of approximately $\pi M \ln M/(8h_D\sqrt{D})$. That is, we can expect to find a prime p after a total number of trials of (v, w) of some $c(\pi h_D \ln M)/\sqrt{D}$, for some constant c. We search for our p in specific ranges $[S, T]$. Thus our expected number of searches is $\tilde{N}_p = \tilde{N}_p(D, S, T) = c(\pi h_D \ln T - S))/\sqrt{D}$. Our experimental data confirms this, see Tables 6.1, 6.2, 6.3 where for example $[S, T] = [2^{191}, 2^{192}]$.

6.5 Constructing Class Polynomials

Although there are different methods to calculate class polynomials, we adopt that of [2], see also [7]. Let $D = b^2 - 4ac$ be the discriminant of a quadratic form

$$f(x, y) = ax^2 + bxy + cy^2$$

where a, b, c are integers. The quadratic form, $f(x, y)$ is commonly represented by the compact notation $[a, b, c]$. If the integers a, b, c have no common factor, then the quadratic form $[a, b, c]$ is called *primitive*. There are infinitely many quadratic forms of any possible discriminant. We reduce to a finite number by demanding that a root of $f(x, 1)$ lie in a certain region of the complex plane. Let the primitive quadratic form $[a, b, c]$ be of negative discriminant. Let τ be the root of $f(x, 1)$ which lies in the upper half-plane:

$$\tau = (-b + \sqrt{D})/2a.$$

The $[a, b, c]$ is a *reduced form* if τ has complex norm greater than or equal to 1, and $\Re(\tau) \in [-1/2, 1/2)$. Given a discriminant $D < 0$, we can easily find all of the reduced quadratic forms of discriminant D. We then compute the class polynomial

$H_D(x)$ which is the minimal polynomial of the $j(\tau)$. For each value of τ, the j-value (denoted j_i below) is computed as follows:

$$j(\tau) = (256f(\tau) + 1)^3/f(\tau)$$

where

$$f(\tau) = \Delta(2\tau)/\Delta(\tau),$$

$$\Delta(\tau) = q \cdot [1 + \sum_{n\geq 1}(-1)^n(q^{3n(n+1/2} + q^{3n(n-1/2)})]^{24},$$

and

$$q = e^{2\pi i\tau}.$$

Finally, the class polynomial can be constructed by using the following formula:

$$H_D(x) = \prod_{i=1}^{h}(x - j_i)$$

where h is the number of the reduced forms of D, commonly known as the *class number* of D. Since $H_D(x)$ has integer coefficients, it suffices to use sufficient accuracy during the computations so as to determine these coefficients to within $1/2$.

Our approach, as stated earlier, is to construct class polynomials beforehand for given D values. We do this using some software tool specialized for mathematical calculations. In our implementation, we use Maple. Following [2], we set the precision for floating point arithmetic as follows:

$$precision = 10 + \left(\begin{array}{c} h \\ \lfloor h/2 \rfloor \end{array} \right) \cdot \pi\sqrt{|D|} \cdot \sum_{i=1}^{h} 1/a_i,$$

$$N = 10 + \left(\begin{array}{c} h \\ \lfloor h/2 \rfloor \end{array} \right) \cdot \sum_{i=1}^{h} 1/a_i.$$

Here N gives the number of terms to keep in the calculations involving the various $\Delta(\tau)$.

As stated earlier, different methods than the basic use of the j-function applied here can be employed to construct class polynomials. In each of these, one obtains

some class-invariant polynomial for the CM discriminant D. One advantage of using different methods would be to have class polynomials with relatively small integer coefficients. This can be useful to store the coefficients when the processor has limited memory.

6.6 Implementation Results

We implemented the new algorithm using the NTL number theory and algebra package on a Pentium II/450 Mhz based PC. We restricted to $t = 2v + 1$ and $s = 2w + 1$ where $v, w \in Z$. Thus, the prime numbers found in this setting are of the form

$$p = v^2 + v + (w^2 + w)D + \frac{D+1}{4} \tag{6.9}$$

where D satisfies

$$D \equiv 3 \pmod 4.$$

And also, D is chosen such that $(D + 1)/4$ is odd, hence p is odd for any choice of v and w. We obtained the average times to find the prime p and prime u and to calculate the corresponding curve for the following values of D. If u is a nearly prime number then the search time for an admissible pair decreases.

For

$$\mathcal{D} = \{\, 163,\ 403,\ 883 \},$$

the corresponding class polynomials are given in the following:

$$H_{163}(x) = x + 640320;$$

$$H_{403}(x) = x^2 - 1088442034024910558330880000000\, x +$$

$$24528113892293313919795200000;$$

$$H_{883}(x) = x^3 + 167990285381627318187575520800123387904000000000\,x^2$$

$$-151960111125245282033875619529124478976000000\,x$$

$$+34903934341011819039224295011933392896000.$$

We obtained efficiency results for these three cases. When the class number is one, the class polynomial is of degree one; hence the root is obtained without any computation. For the two other cases, we need to find a root of degree 2 and degree 3 polynomials, respectively. The results are given in Table 6.1.

Table 6.1. Timings to build curves of known order.

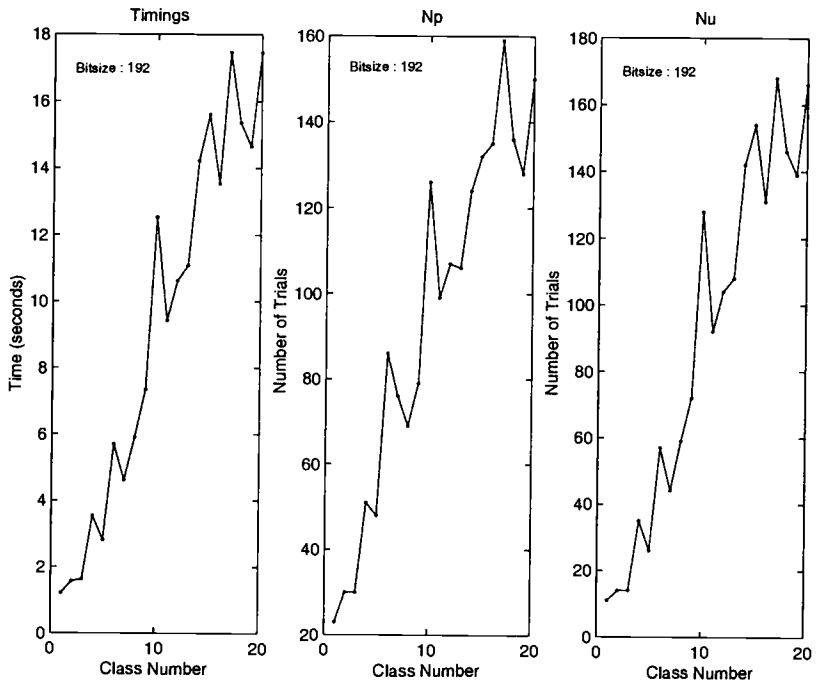
D	class no	bitsize	Average time (s)	N_p	N_u
163	1	192	1.22	23	11
163	1	224	2.29	27	14
403	2	192	1.57	30	14
403	2	224	3.29	36	21
883	3	192	1.63	30	14
883	3	224	3.01	36	19

To find a root of a class polynomial of modulo p takes approximately a constant time determined by the size of the modulus p and the degree of the polynomial. However, the time or the number of trials to find admissible pairs of p and u is of a more complicated nature. We have run our program repeatedly to build 1000 different curves with each value of D in Table 6.1. Although their probabilistic properties are not known explicitly, we observed that expected values of time and number of trials seemed to remain constant over different runs of the program. In the table, N_p indicates the approximate number of random pairs of v and w to be tried before a prime $p = v^2 + v + (w^2 + w)D + (D+1)/4$ is found. Similarly, N_u is the average trial number of p of the form (6.9) to obtain a prime u.

Table 6.2. Timings to build curves of known order.

bitsize	D	class no	Average time (s)	N_p	N_u
192	555	4	3.54	51	35
	1051	5	2.78	48	26
	451	6	5.70	86	57
	811	7	4.61	76	44
	1299	8	5.91	69	59
	1187	9	7.35	79	72
	611	10	12.53	126	128
	1283	11	9.42	99	92
	1235	12	10.62	107	104
	1451	13	11.08	106	108
	1211	14	14.22	124	142
	1259	15	15.61	132	154
	1379	16	13.54	135	131
	1091	17	17.46	159	168
	1691	18	15.35	136	146
	2099	19	14.64	128	139
	1739	20	17.45	150	166

Figure 6.1. Performance of the method with increasing class numbers.



To show that the method is still efficient for larger class numbers we supplied numerical results for the values of D with each class number in the interval $[1, 20]$ in Table 6.2 and Figure 6.1. As can be observed easily from the figure the time needed to find admissible pairs increases as the class number becomes bigger. Although this increase is not monotone — the timing for class number 10 is much higher than those for class numbers 11, 12, and 13 — it is reasonable to claim that the time needed to find proper pairs is directly proportional to the class number. This result is consistent with the theoretical considerations in [17]; see the previous section for further comments. The dependence of the construction process on the particular value of D seems to account for the deviation from simple monoticity. Note also, just as the theoretical heuristics of the previous section suggest, that the time to find an admissible pair (p, u) decreases as the value of D increases. This can be observed in Tables 6.3 and 6.4, and Figures 6.2 and 6.5.

Table 6.3. Timings for different D values for certain classes with degree 192.

bitsize	class no	D	Average time (s)	N_p	N_u
192	1	11	9.10	95	94
		19	3.86	68	39
		43	2.30	46	23
		67	1.87	37	18
		163	1.22	23	11
	2	35	10.38	105	108
		123	3.49	57	35
		187	2.42	45	23
		235	2.09	40	20
		403	1.57	30	14
	3	59	11.37	121	118
		83	10.01	102	104
		107	7.90	92	82
		379	2.63	47	25
		883	1.63	30	14
	4	155	9.50	99	99
		195	6.46	88	66
		259	4.77	78	49
		355	3.76	64	37
		555	3.54	51	35
	5	179	11.54	113	119
		227	9.33	103	97
		347	7.64	83	79
		443	6.65	73	68
		1051	2.78	48	26

Table 6.4. Timings for different D values for certain classes with degree 224

bitsize	class no	D	Average time (s)	N_p	N_u
224	1	11	16.20	109	113
		19	7.15	81	49
		43	4.19	55	28
		67	3.55	44	23
		163	2.29	27	14
	2	35	15.74	120	110
		123	5.93	64	40
		187	4.31	52	28
		235	3.98	48	26
		403	3.29	36	21
	3	59	21.17	141	128
		83	16.93	118	117
		107	14.33	106	99
		379	4.85	56	32
		883	3.01	36	19
	4	155	16.14	116	112
		195	11.90	105	82
		259	8.46	91	58
		355	6.87	77	46
		555	6.54	63	44
	5	179	20.65	140	142
		227	17.42	122	120
		347	12.64	98	86
		443	11.81	86	81
		1051	5.52	55	36

Figure 6.2. Timings to build curves with increasing discriminants.

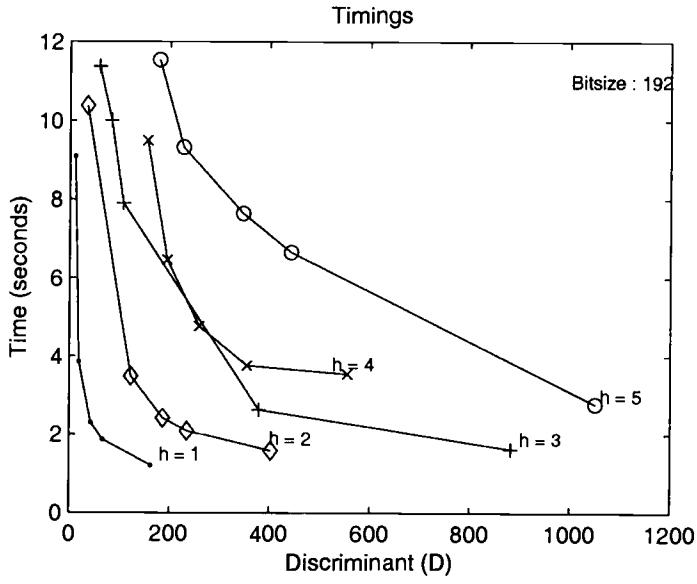


Figure 6.3. Number of trials for p with increasing discriminants.

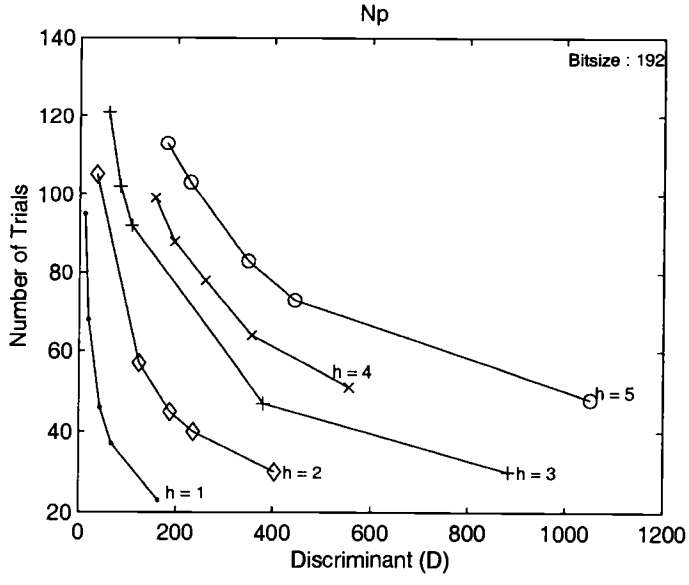


Figure 6.4. Number of trials for u with increasing discriminants.

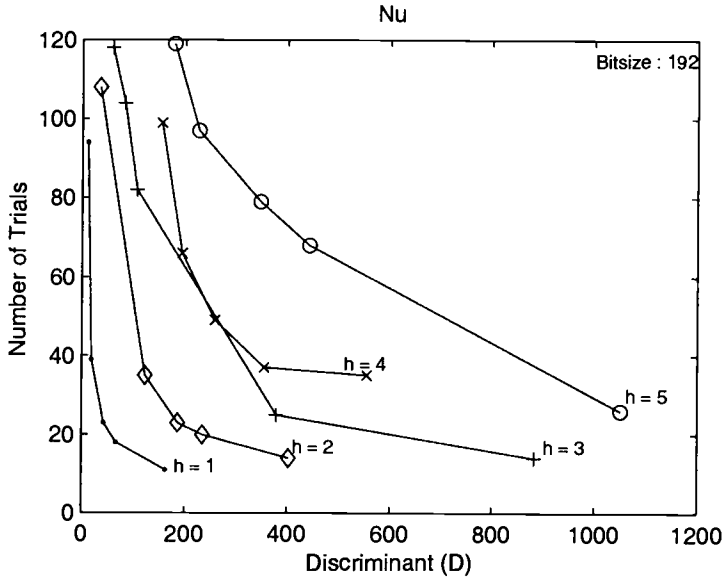


Figure 6.5. Timings to build curves with increasing discriminants.

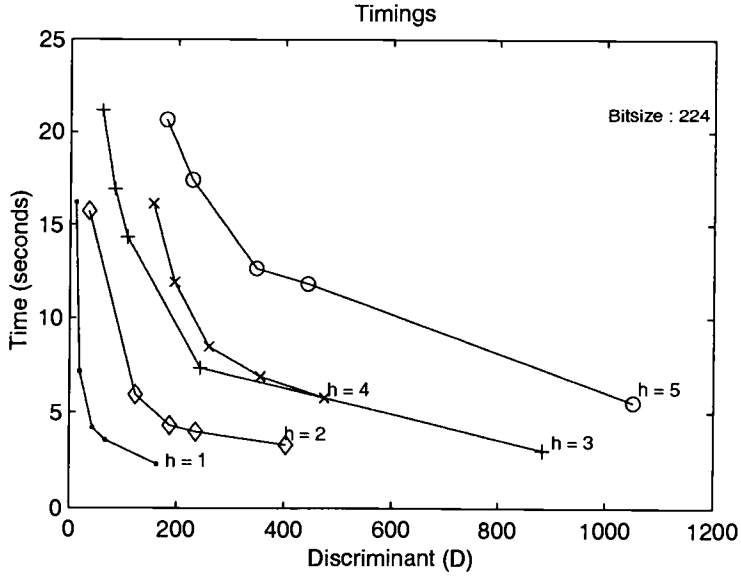


Figure 6.6. Number of trials for p with increasing discriminants.

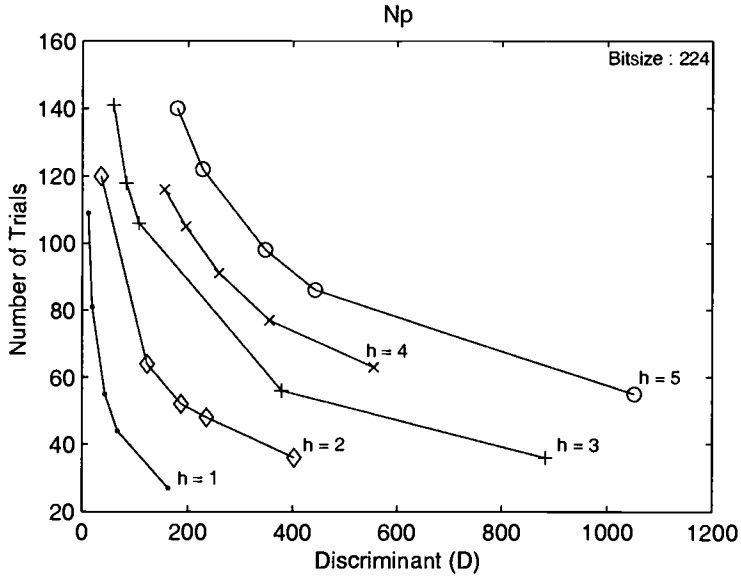
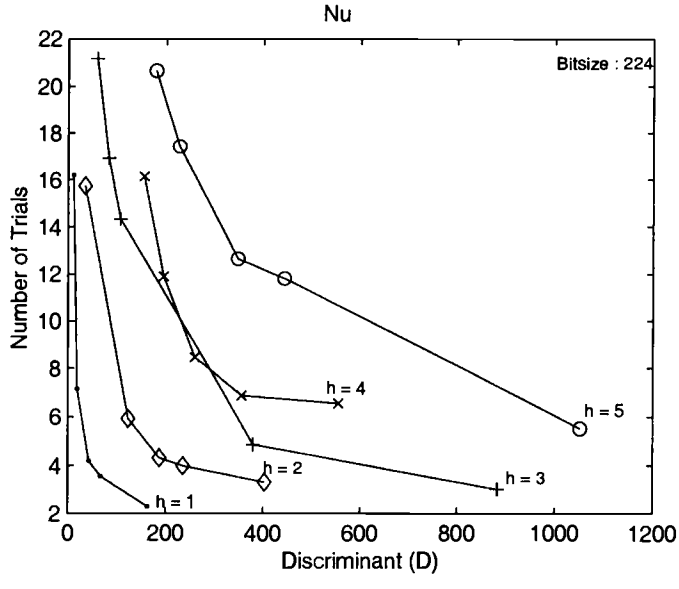


Figure 6.7. Number of trials for u with increasing discriminants.



Another important implementation aspect is code size. While one implementation [49] of the full CM method [15] requires 204KB on a PC with WindowsNT, our implementation with NTL requires only 164KB code space on the same platform. In fact, the code space can be made much smaller when code is written expressly for curve generation. For sake of simplicity, we have written such a program which treats only the class number one case. We found that only an extra 10 KB of object code space is needed for curve generation routines (assuming that the basic subroutines for arithmetic operations needed for elliptic curve arithmetic are already available).

6.7 Conclusion

We present a variant of the complex multiplication (CM) elliptic curve generation algorithm for $GF(p)$. We modified the existing CM algorithm on the assumption that the field $GF(p)$ is not fixed and we have the flexibility of selecting it at random from certain subsets of prime fields. We show that the new variant of the CM method provides smaller, faster and more easily coded software implementation. The modified algorithm utilizes a collection of precomputed class polynomials of relatively low degree corresponding to a predetermined set of CM discriminants. In fact, the characteristic p of the field $GF(p)$ that can be given as a function of a CM discriminant is a member of a large subset of prime numbers. The theoretical analysis shows that there are numerous prime numbers in this subset and experimental results confirm that it is highly probable to cosntruct a prime number belonging to this set with a fairly small number of searches. Our experiments also reveal the fact that the performance of the modified CM method increases as the class number decreases. Another interesting result is that the new CM method perfoms better for larger discriminants of the same class.

Chapter 7
Conclusion

7.1 Summary of the Contributions

This thesis proposes practical solutions for various implementation problems encountered in elliptic curve cryptography. We can group these implementation problems considered here into three categories:

- Problems related to elliptic curve implementations using Montgomery arithmetic.
- Problems related to composite field arithmetic (i.e. the arithmetic in binary extension fields with composite exponents).
- Fast and simple software implementations for elliptic curve generation.

The first category can be treated in two subcategories: Montgomery inversion and Montgomery multiplication.

The use of Montgomery multiplication algorithm in elliptic curve cryptosystems necessitates an efficient implementation of Montgomery modular inverse operation. A faster Montgomery inverse implementation is beneficial especially in elliptic curve cryptosytems employing affine coordinates. We proposed three algorithms for computing the classical modular inverse, the Kaliski-Montgomery inverse, and the new Montgomery inverse. The proposed algorithms are suitable for software implementations on general-purpose microprocessor. The speedups obtained with these three algorithms over the classical algorithms are between 1.14 and 1.56 in the range of interest for elliptic curve cryptography. The classical approach requires a bidirectional conversion operations between the residue (non-Montgomery) domain and

the Montgomery domain every time when an inversion operation is necessary since it uses the standard modular arithmetic. Therefore, we emphasize the importance of using Montgomery arithmetic, which eliminates the conversion during elliptic curve operations.

Majority of elliptic curve implementations utilize two types of fields: binary extension fields $GF(2^m)$ and prime fields $GF(p)$. When the speed is important, hardware implementations of modular multiplication in both fields become necessary. There are hardware implementations which are very efficient especially for binary extension fields with certain types of trinomials and for prime fields with certain forms of characteristics. These types of designs can not be used for any other field than the intended one. We proposed a unified design that operates in both fields as well as with any prime characterictic and any irreducible polynomial. The multiplier is scalable in the sense that it can be reused or replicated in order to function with longer operand precisions independently of the data path precision for which the unit was originally designed. Additionally, the design allows the wordsize to be selected based on the area and performance requirements. The proposed multipler was synthesized using Mentor tools, and a circuit capable of working with clock frequencies up to 90 MHz was obtained. We observed that the speedup of the hardware module over a software implementation on a comparable microprocessor with the same clock frequency is between 4.46 and 5.02 in the range of interest for elliptic curve cryptography. The speedup for larger precisions such as 1024 bits, which is a typical wordsize for RSA cryptosystem, attains 9.34.

In the second category, we deal with the arithmetic in binary extension fields with composite exponents. Although certain kinds of efficient attacks on elliptic curve cryptosystems using composite fields have been revealed recently, the composite fields still stands as a good choice for software implementation of other types of cryptographic systems such as hyperelliptic curve cryptosystems due to their word-oriented arithmetic.

The efficiency of the composite field arithmetic on general-purpose microprocessors closely depends on the choice of basis and the composition of the field degree. We present performance results of implementations using three different bases: polynomial base, optimal normal base type I, and type II. There are basically three arithmetic operations for which we need efficient algorithms: multiplication, squaring and multiplicative inversion. In addition to determining the best previously known algorithm for some operations in a field, we also proposed novel algorithms for the others.

We also addressed the conversion problem between a composite field and a binary field since conversion must be performed when one of two communicating parties uses composite field and the other uses binary field. We defined the method to construct the conversion matrices between the two fields. In addition to that, we also presented storage efficient conversion algorithms when the conversion matrices are too large to store in computer memory.

In the last category, we addressed a very difficult problem: elliptic curve generation. We made certain simplification to a previously suggested algorithm in order to implement it in a small microprocessor with better time efficiency of constructing a suitable curve. We also presented the observations on the plentitude of suitable primes and elliptic curves to demonstrate the efficiency of the new method.

7.2 Directions for Future Research

In this thesis, two concepts regarding the elliptic curve implementations have been given special attention:

- scalability
- dual-field arithmetic in unified architectures

A scalable and unified multiplier architecture for both types of finite fields $GF(p)$ and $GF(2^m)$ was defined in this thesis. The architecture is capable of performing multiplication in $GF(p)$ and $GF(2^m)$ for any operand precision. However,

the register lengths for operands and intermediary results vary with the precision. It is possible to calculate register lengths in advance by fixing an upper bound for the operand precision. In this case, the module wil not be able to work with precisions bigger than the fixed value. Therefore, this feature of the design contradicts with full scalability precept. In this thesis, the registers are not considered the essential parts of the architecture and treated as external components. Limiting the length of the registers within the multiplier module and using an external memory for excessive words of the operands (or of intermediary values) was proposed as a solution. It was also mentioned that this solution might lead to performance degradation due to possible pipeline stalls when the memory access and transfer rate are not sufficient. In order to prevent pipeline stalls, special memory access mechanisms must be empoyed. But the length of internal registers remain to be determined in accordance with the application.

In elliptic curve cryptography, one of the most important operation is the multiplicative inverse operation in the finite field employed. Especially, when the affine coordinate system is employed, the inverse turns out to be dominant operation in terms of time. A unified architecture for performing inverse operation in both finite field $GF(2^m)$ and $GF(p)$ is extremely beneficial. Indeed, it is possible to redefine two known algorithms – the "Almost Inverse Algorithm" for $GF(2^m)$ and "Montgomery Modular Inverse Algorithm" for $GF(p)$ – in a way that the steps of the two algorithms are similar. Furthermore, the architecture must also be scalable to larger operand precisions. A fast, unified and scalable inversion module might even remove the necessity of using projective coordinates, which might prevent a hardware implementation of elliptic curve cryptosystems due to large amounts of memory requirements in projective coordinates.

BIBLIOGRAPHY

[1] G. B. Agnew, R. C. Mullin, and S. A. Vanstone. An implementation of elliptic curve cryptosystems over $F_{2^{155}}$. *IEEE Journal on Selected Areas in Communications*, 11(5):804–813, June 1993.

[2] A. O. L. Atkin and F. Morain. Elliptic curves and primality proving (english summary). *Math. Comp.*, (61) 126(203):29–68, 1993.

[3] A. Bernal and A. Guyot. Design of a modular multiplier based on Montgomery's algorithm. In *13th Conference on Design of Circuits and Integrated Systems*, pages 680–685, Madrid, Spain, November 17–20 1998.

[4] J. V. Brawley and G. E. Schnibben. *Infinite Algebraic Extensions of Finite Fields.* American Mathematical Society, Providence, RI, 1989.

[5] H. Cohen. *A course in computational algebraic number theory. Graduate Texts in Mathematics, 138.* Springer-Verlag, Berlin, 1997.

[6] H. Cohn. *Advanced Number Theory.* Dover, 1980.

[7] David A. Cox. *Primes of the form x^2+ny^2. Fermat, class field theory and complex multiplication. A Wiley-Interscience Publication. A Wiley-Interscience Publication.* John Wiley and Sons,Inc., New York, 1989.

[8] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, November 1976.

[9] Ö. Eğecioğlu and Ç. K. Koç. Exponentiation using canonical recoding. *Theoretical Computer Science*, 129(2):407–417, 1994.

[10] S. E. Eldridge and C. D. Walter. Hardware implementation of Montgomery's modular multiplication algorithm. *IEEE Transactions on Computers*, 42(6):693 699, June 1993.

[11] Steve Furber. *ARM System Architecture.* Addison-Wesley, Reading, MA, 1997.

[12] J. Guajardo and C. Paar. Efficient algorithms for elliptic curve cryptosystems. In B. S. Kaliski Jr., editor, *Advances in Cryptology — CRYPTO 97*, Lecture Notes in Computer Science, No. 1294, pages 342–356. Springer, Berlin, Germany, 1997.

[13] G. Harper, A. Menezes, and S. Vanstone. Public-key cryptosystems with very small key lengths. In R. A. Rueppel, editor, *Advances in Cryptology — EUROCRYPT 92*, Lecture Notes in Computer Science, No. 658, pages 163–173. Springer, Berlin, Germany, 1992.

[14] IEEE P1363. Standard specifications for public-key cryptography. Draft Version 7, September 1998.

[15] IEEE P1363. Standard specifications for public-key cryptography. Draft Version 13, November 12, 1999.

[16] T. Itoh and S. Tsujii. A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases. *Information and Computation*, 78(3):171–177, September 1988.

[17] H. W. Lenstra Jr. Factoring integers with elliptic curves. *Annals of Mathematics*, (2) 126(3):649–673, 1987.

[18] B. S. Kaliski Jr. The Montgomery inverse and its applications. *IEEE Transactions on Computers*, 44(8):1064–1065, August 1995.

[19] B. S. Kaliski Jr. and M. Liskov. Efficient finite field basis conversion involving dual bases. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems*, Lecture Notes in Computer Science, No. 1717, pages 135–143. Springer, Berlin, Germany, 1999.

[20] B. S. Kaliski Jr. and Y. L. Yin. Methods and apparatues for efficient finite field conversion. U.S. Patent Number 5,854,759, December 29, 1998.

[21] B. S. Kaliski Jr. and Y. L. Yin. Storage-efficient finite field basis conversion. In S. Tavares and H. Meijer, editors, *Selected Areas in Cryptography*, Lecture Notes in Computer Science, No. 1556, pages 81–93. Springer, Berlin, Germany, 1998.

[22] D. E. Knuth. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms.* Addison-Wesley, Reading, MA, Third edition, 1998.

[23] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, January 1987.

[24] Ç. K. Koç. High-Speed RSA Implementation. Technical Report TR 201, RSA Laboratories, 73 pages, November 1994.

[25] Ç. K. Koç and T. Acar. Montgomery multiplication in GF(2^k). *Designs, Codes and Cryptography*, 14(1):57–69, April 1998.

[26] Ç. K. Koç, T. Acar, and B. S. Kaliski Jr. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.

[27] Ç. K. Koç and B. Sunar. Low-complexity bit-parallel canonical and normal basis multipliers for a class of finite fields. *IEEE Transactions on Computers*, 47(3):353–356, March 1998.

[28] P. Kornerup. High-radix modular multiplication for cryptosystems. In E. Swartzlander, Jr., M. J. Irwin, and G. Jullien, editors, *Proceedings, 11th Symposium on Computer Arithmetic*, pages 277–283, Windsor, Ontario, June 29 – July 2 1993. IEEE Computer Society Press, Los Alamitos, CA.

[29] R. Lidl and H. Niederreiter. *Introduction to Finite Fields and Their Applications.* Cambridge University Press, New York, NY, 1994.

[30] A. J. Menezes. *Elliptic Curve Public Key Cryptosystems.* Kluwer Academic Publishers, Boston, MA, 1993.

[31] A. J. Menezes, I. F. Blake, X. Gao, R. C. Mullen, S. A. Vanstone, and T. Yaghoobian. *Applications of Finite Fields.* Kluwer Academic Publishers, Boston, MA, 1993.

[32] V. Miller. Uses of elliptic curves in cryptography. In H. C. Williams, editor, *Advances in Cryptology — CRYPTO 85, Proceedings*, Lecture Notes in Computer Science, No. 218, pages 417–426. Springer, Berlin, Germany, 1985.

[33] A. Miyaji. Elliptic curves over F_p suitable for cryptosystems. In *Advances in Cryptology — AUSCRYPT' 92*, Lecture Notes in Computer Science, No. 718, pages 479–491. Springer-Verlag, Berlin, 1993.

[34] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[35] D. Naccache and D. M'Raïhi. Cryptographic smart cards. *IEEE Micro*, 16(3):14–24, June 1996.

[36] National Institute for Standards and Technology. Digital signature standard (DSS). Federal Register, 56:169, August 1991.

[37] National Institute for Standards and Technology. Digital Signature Standard (DSS). FIPS PUB 186-2, January 2000.

[38] J.-H. Oh and S.-J. Moon. Modular multiplication method. *IEE Proceedings: Computers and Digital Techniques*, 145(4):317–318, July 1998.

[39] J. Omura and J. Massey. Computational method and apparatus for finite field arithmetic. U.S. Patent Number 4,587,627, May 1986.

[40] H. Orup. Simplifying quotient determination in high-radix modular multiplication. In S. Knowles and W. H. McAllister, editors, *Proceedings, 12th Symposium on Computer Arithmetic*, pages 193–199, Bath, England, July 19–21 1995. IEEE Computer Society Press, Los Alamitos, CA.

[41] C. Paar. *Efficient VLSI Architectures for Bit Parallel Computation in Galois Fields.* PhD thesis, Universität GH Essen, VDI Verlag, 1994.

[42] C. Paar, P. Fleischmann, and P. Soria-Rodriguez. Fast arithmetic for public-key algorithms in Galois fields with composite exponents. *IEEE Transactions on Computers*, 48(10):1025–1034, October 1999.

[43] C. Paar and P. Soria-Rodriguez. Fast arithmetic architectures for public-key algorithms over Galois fields $GF((2^n)^m)$. In W. Fumy, editor, *Advances in Cryptology — EUROCRYPT 97*, Lecture Notes in Computer Science, No. 1233, pages 363–378. Springer, Berlin, Germany, 1997.

[44] J.-J. Quisquater and C. Couvreur. Fast decipherment algorithm for RSA public-key cryptosystem. *Electronics Letters*, 18(21):905–907, October 1982.

[45] M. Rosing. *Implementing Elliptic Curve Cryptography.* Manning Publications, Greenwich, CT, 1999.

[46] A. Royo, J. Moran, and J. C. Lopez. Design and implementation of a coprocessor for cryptography applications. In *European Design and Test Conference*, pages 213–217, Paris, France, March 17-20 1997.

[47] E. Savaş and Ç. K. Koç. The Montgomery modular inverse - revisited. *IEEE Transactions on Computers*, 49(8), July 2000. To appear.

[48] R. Schroeppel, H. Orman, S. O'Malley, and O. Spatscheck. Fast key exchange with elliptic curve systems. In D. Coppersmith, editor, *Advances in Cryptology — CRYPTO 95*, Lecture Notes in Computer Science, No. 973, pages 43–56. Springer, Berlin, Germany, 1995.

[49] M. Scot. A C++ implementation of the complex multiplication (CM) elliptic curve generation algorithm from Annex A by Mike Scott. ftp://ftp/compapp.dcu.ie/crypto/cm.cpp, 1999.

[50] I. A. Semaev. Evaluation of discrete logarithms in a group of p-torsion points of an elliptic curve in characteristic p. *Math. Comp.*, 67(221):353–356, 1998.

[51] G. Seroussi. Table of low-weight binary irreducible polynomials. Hewlett-Packard, HPL-98-135, August 1998.

[52] J. H. Silverman. Fast multiplication in finite field $GF(2^N)$. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems*, Lecture Notes in Computer Science, No. 1717, pages 122–134. Springer, Berlin, Germany, 1999.

[53] Joseph H. Silverman. *The arithmetic of elliptic curves. Corrected reprint of the 1986 original. Graduate Text in Mathematics, 106.* Springer-Verlag, New York, 1997.

[54] B. Sunar and Ç. K. Koç. An efficient optimal normal basis type II multiplier, January 1999. Submitted for publication.

[55] A. F. Tenca and Ç. K. Koç. A scalable architecture for Montgomery multiplication. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems*, Lecture Notes in Computer Science, No. 1717, pages 94–108. Springer, Berlin, Germany, 1999.

[56] C. D. Walter. Space/Time trade-offs for higher radix modular multiplication using repeated addition. *IEEE Transactions on Computers*, 46(2):139–141, February 1997.

[57] E. De Win, A. Bosselaers, S. Vandenberghe, P. De Gersem, and J. Vandewalle. A fast software implementation for arithmetic operations in GF(2^n). In K. Kim and T. Matsumoto, editors, *Advances in Cryptology — ASIACRYPT 96*, Lecture Notes in Computer Science, No. 1163, pages 65–76. Springer, Berlin, Germany, 1996.

[58] H. Wu. Low complexity bit-parallel finite field arithmetic using polynomial basis. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems*, Lecture Notes in Computer Science, No. 1717, pages 280–291. Springer, Berlin, Germany, 1999.

[59] H. Wu, M. A. Hasan, and I. F. Blake. Highly regular architectures for finite field computation using redundant basis. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems*, Lecture Notes in Computer Science, No. 1717, pages 269–279. Springer, Berlin, Germany, 1999.